pBseq: probabilistic buffer-swapping sequence locks

Nicholas Mc Guire <safety@osadl.org>

October 21, 2016

Open Source Automation Development Lab eG

Outline

- Context - seqlocks and RT
- Alternatives
- pBseq
- Entropy harvesting
- Conclusions

Nicholas Mc Guire <safety@osadl.

sequence locks overview

```
single writer:

  data1 = ...;
  data2 = ...;
  seq = seq + 2;
```

```
multiple readers:

unsigned seq_start, seq_end;
do {
        seq_start = seq;
        local1 = data1;
        local2 = data2;
        seq_end = seq;
} while (seq_start != seq_end
          || seq_start % 2);
```

sequence locks properties

- Writer never blocks on reader (no writer starvation)
- Reader may spin on writer (but no reader starvation)
- Multiple readers permitted
- Single writer (multi-writer -> by locks)
- Writers preferred over readers

Sequence locks have been in use since the late 2.5 kernel series,

The seqlock problems for RT

- Reader retries are not bounded - even though generally short
- Anonymous locks do not permit boosting
- Readers/writers can't be boosted -> unbounded delays on writer preemption
- Long spinning readers -> cache impact

Current mitigation

- Basic options
  - Try to add owner concept to sequence locks
  - Revert to boostable locks
- 1st option would be complex and unnecessary for non-RT
- 2nd is simple but limits scalability of RT
  - write_seqlock: grabs the seqlock spin_lock (multiple writer serialization)
  - read_seqbegin: spin_lock(&sl->lock); spin_unlock(&sl->lock); so it is bostable now.

No real mitigation in the current RT patch-set - the its more a workaround.

Nicholas Mc Guire
<safety@osadl.

Alternative mitigation: replication

- SeqLocks are unbounded due to possible writer preemption
- Mitigation: replicated data
- Implementations:
  - seqcount_latch (fast ktime and in latch_tree_OP)
  - Suitable for non-atomic modifications
  - Prime motivation: unconditional lookups e.g. NMI context

For the NMI case data duplication is sufficient. This also could solve some of the RT issues but its not a simple replacement (code-level) for seqlocks.

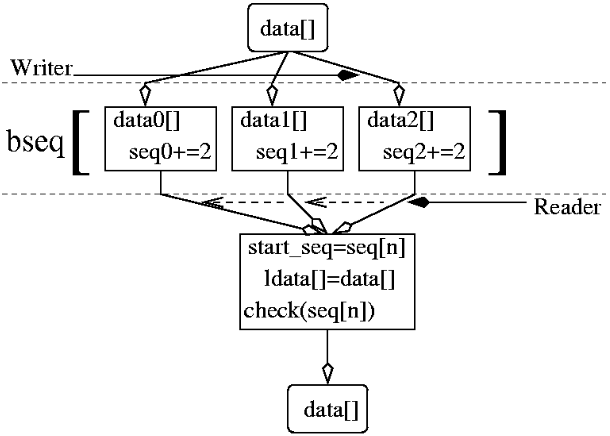
Concept of seqcount]_latch

OSADL

- Maintain replicated data - data[0]/data[1]
- Redirect readers to stable copy
- Use LSB of sequence to select data buffer
- Non-probabilistic approach - redirection is deterministic
- Retry probability is assumed to be negligible due to preemption
- Worst-case: still unbounded - lockstep update/read possible

Lockstep behavior on larger multi-core is actually possible since cross-core delays can be quite large.

pBseq probabilistic buffer-swapping sequence locks - overview



Concept of pBseq

- Maintain replicated data - data[NUM_REPLICA]
- Probabilistic approach - redirect readers to random copy
- Use inherent nondeterminism to select data copy
- Retry probability is statistically bounded even in the tight-loop
- Data copying on the read-side - its more of an IPC than a lock
- Worst-case: bounded - no lockstep update/read

pBseq parameters

- Number of replicas
- Data array size
- Treatment of index
  - -> collision avoidance (seqcount_latch)
  - -> lockstep avoidance
- Memory model of architecture
  - TSO: no rmb()/wmb() needed (no shared writable data)
  - non-TSO: barriers needed

Implementation on TSO systems

Writer:

```
for (b = 0; b < NUM_REP; b++) {
  p = &seq.bseq[b];
  p->s++;
  for (d = 0; d < D_SIZE; d++){
    randbit=(~randbit);
    p->data[d] = ...;
    randbit=(~randbit);
  }
  p->s++;
}
```

Reader:

```
static unsigned int idx = 1;
idx -= randbit;
do {
  lp = &seq.b[(--idx)%NUM_REP];
  lseq = lp->s;
  for (d = 0; d < D_SIZE; d++){
    ldata[d] = lp->data[d];
  }
  randbit=(~randbit);
} while (lseq%2 || lseq!=lp->s);
```

Implementation on non-TSO (ARM/PPC)

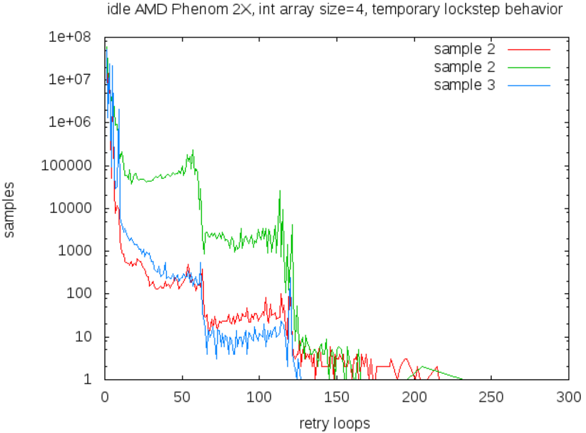
OSADL

- Failure rate is actually quite small (10E-7/call)
- Memory barriers mandatory and expensive
- Works - but no systematic testing yet
- Performance issues and optimization open
- Unclear if a barrier free version is actually possible

Not done yet - the distributions look more or less the same, but there are rare failures if there is a barrier free mitigation is not yet clear - still working on basics here...

The retry lockstep problem

- Temporary lockstep behavior possible
- Even short lock-stepping could have significant cache side-effects

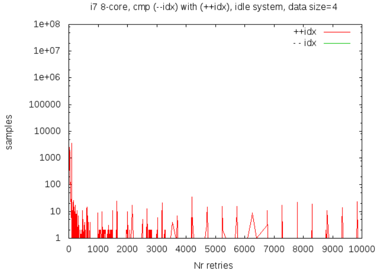


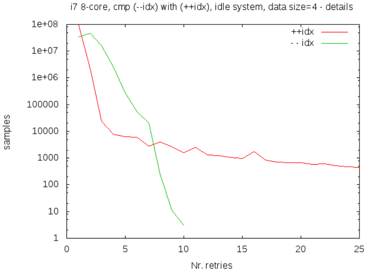
Mitigations solution-space

- Defined writer start conditions and inverted read access direction
- Defined writer start conditions and fixed offsets (seqcount_latch)
- Use of random index (symmetry-breaking)
- Larger replica arrays
- Execution interleaving
- System load

...and any combination of the above

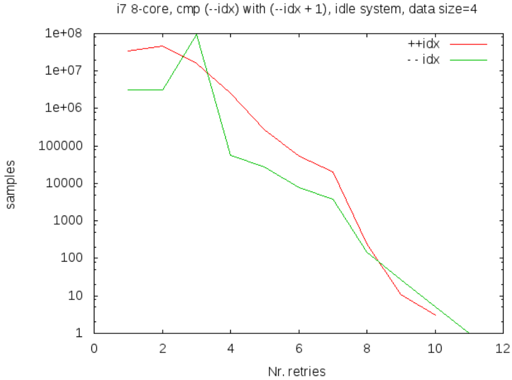
Defined access direction comparison





The distribution indicates that there are very long lockstep sequences possible in tight-loop runs.

Defined index offset comparison



- unsigned int idx=1; -> lp = &seq.bseq[(–idx)
- unsigned int idx=0; -> lp = &seq.bseq[(–idx + 1)

Non-determinism

- Explicit random numbers
  - Possible but too expensive
- Utilize the asynchrony of system
  - Non-synchronized race on global var
- Utilize the history of the system
  - Static declaration index variable

A reliable entropy source suitable for low-level algorithms to ensure symetry breaking is not only usable for pBseq/pWCS but is a pre-requisite to bring probabilistic solutions to low-level algorithms in general.

Entropy harvesting - randbits

Core of the essed.c

```
unsigned int coin;
```

Thread A

```
  while(active){
    coin=~coin;
```

Thread B

```
  unsigned int draw[2];
  drwa[0]=coin;
  draw[1]=coin;
  if(draw[0]<draw[1])
    inbuf|=(1<<pos);
  else
    inbuf&=~(1<<pos);
  pos++;
```

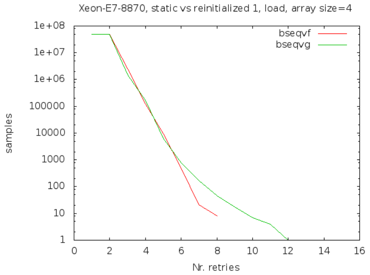
Entropy harvesting - state history

OSADL

```
<       static unsigned int idx=1;
---
>       unsigned int idx=1;
```

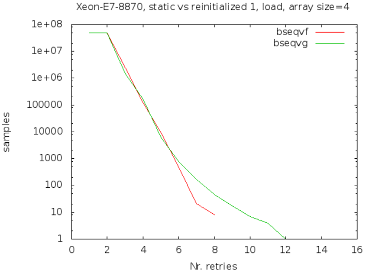
Entropy harvesting - state history

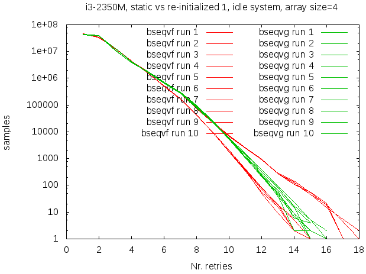
```
<       static unsigned int idx=1;
---
>       unsigned int idx=1;
```



Entropy harvesting - state history

```
<       static unsigned int idx=1;
---
>       unsigned int idx=1;
```

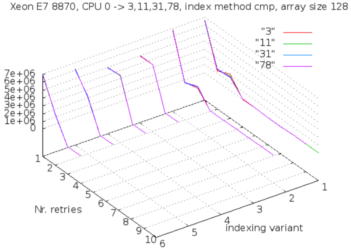




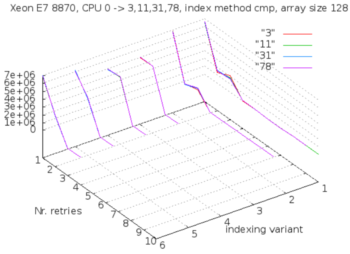
Entropy insertion points

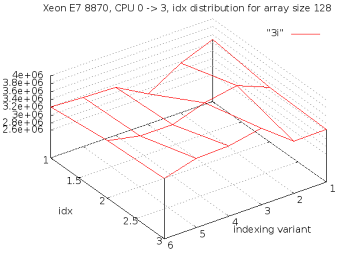

```
int idx <----- idx = randbit;
        <----- static int idx = 0;
        <----- idx -= randbit;
do {
        lp = &seq.bseq[(--idx)%NUM_REPLICA]; <--- +randbit
        lseq = lp->s;
        for (d = 0; d < DATA_SIZE; d++) {
                ldata[d] = lp->data[d];
                  <--------------------randbit=(~randbit);
        }
          <----------------------------randbit=(~randbit);
} while (lseq%2 || lseq != lp->s);
```

Key problem - where to insert entropy



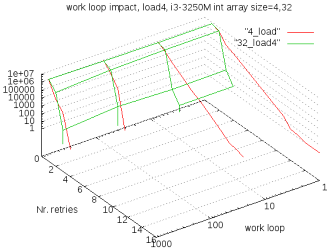
Key problem - where to insert entropy



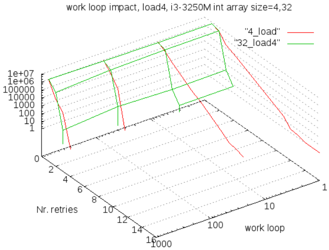


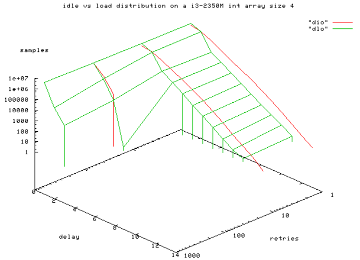
Looking at the retrydistribution in conjunction with the index distribution

Worst-case: minimal delays + idle system



Worst-case: minimal delays + idle system





Exactly one worst case scenario -> exhaustive testing possible ?

Current "best" code

```
static unsigned int idx=1;
for (n = 1; n <= N; ++n) {
        idx += randbit;
        do {
                lp = &seq.bseq[(--idx)%NUM_REPLICA];
                lseq = lp->s;
                for (d = 0; d < DATA_SIZE; d++) {
                        ldata[d] = lp->data[d];
                }
                randbit=(~randbit);
        } while (lseq%2 || lseq != lp->s);
```

Properties of pBseq

- Freshness: as good as spinlocks/mutexes
- Context separation: reader/writer lock-free/wait-free always
- Performance: Statistically bounded retries
- Worst Case:
  - Idle-system
  - Tight-loop
  - Sall data
  - -> exhaustive testable worst case

Nicholas Mc Guire
<safety@osadl.

Conclusions

- Related: pWCS and similarities with sequcount_latch
- pBseq is more of an IPC mechanism than a lock (copying semantics)
- Exhaustive testing of worst-case possible
- Simple inherent entropy harvesting is usable in low-level algorithms
- Symmetry breaking can be implemented in a highly reliable manner -> robust guarantees on retries
- Using inherent non-determinism in a systematic form (code wise) is still an open issue
- The code sensitivity to very small changes is large - making the evaluation of the code very hard.

We think its worth digging deeper - synchronization based on robust statistical properties may be a scalable alternative to deterministic locking.