Adventures in Verification

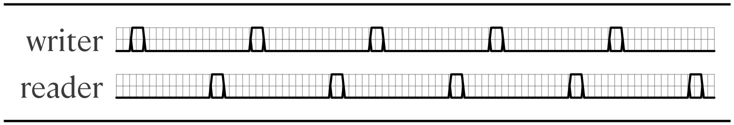
Glorified Ring Buffers

Kai Engelhardt

Ghost Locomotion
Mountain View, CA, USA and Sydney, AU

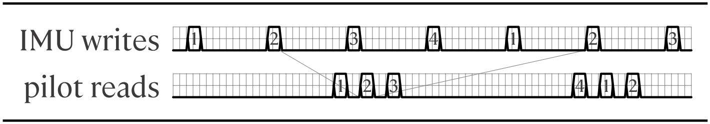
In Synchronous World, Far, Far Away ...

...two programs communicate via shared memory



In Synchronous World, Far, Far Away ...

...an IMU[1] writing values read in 3s from a 4-slot ring buffer

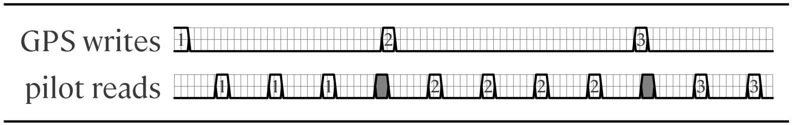


(numbers are ring buffer indices)

[1]Confused by acronym bingo? ▸ Check the glossary.

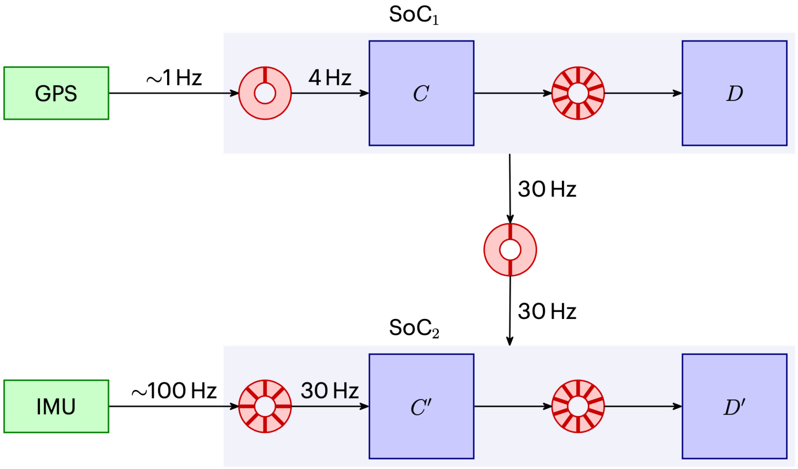
In a Slightly Less Synchronous World…

…some shared memory reads clash with writes
but we make up for it by sampling often enough



(numbers are values written/read to the single shared memory location)

Typical Comm's Problems



Some Requirements for a Comm's Primitive

- store m most recent payloads of size n
- $\mathcal{O}(m)$ memory overhead
- wait-free $\mathcal{O}(n)$ reads and writes of individual payloads
- cannot assume atomic reads/writes of entire payloads

Some Requirements for a Comm's Primitive

- store m most recent payloads of size n
- $\mathcal{O}(m)$ memory overhead
- wait-free $\mathcal{O}(n)$ reads and writes of individual payloads
- cannot assume atomic reads/writes of entire payloads

Problem: Can't have it all! Need to weaken at least one req.

Some Requirements for a Comm's Primitive

- store m most recent payloads of size n
- $\mathcal{O}(m)$ memory overhead
- wait-free $\mathcal{O}(n)$ reads and writes of individual payloads
- cannot assume atomic reads/writes of entire payloads

Problem: Can't have it all! Need to weaken at least one req.

Choice: if reads can fail, at least we can make read failure detectable. We call the data structures GRBs (for *glorified ring buffers*).

C Prototypes of GRB Ops

```
grb_ret_t grb_read(Grb_t *g, size_t i, QID *q, Chunk c[NUMCHUNKS]);

void grb_write(Grb_t *g, size_t i, QID q, Chunk c[NUMCHUNKS]);
```

where

- all data is tagged with a 64-bit `QID` (almost a nonce)
- `grb_read` returns whether the attempted read from slot `i` of GRB `g` into payload buffer `c` and quantum ID (QID) `q` succeeded,
- `grb_write` writes payload buffer `c` and QID `q` into slot `i` of GRB `g`.

GRB Correctness Property

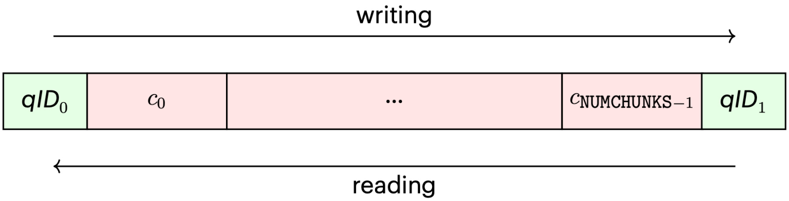
If the reader finds

```
read(&g, i,  &q, c) == GRB_OK
```

then its slot value `(q, c)` equals the `i`'th slot of `g` when it was last written by the writer.

Roughly: the read violation detection works.

Folklore: Lamport's Read-Forward-Write-Backward



GRB Types

```
typedef struct {
  QID q0;
  Chunk c[NUMCHUNKS];
  QID q1;
} Slot_t;

typedef struct {
  Slot_t b[NUMSLOTS];
} Grb_t;
```

GRB Write Operation

```
void grb_write(Grb_t *g, size_t i, QID q, Chunk c[NUMCHUNKS]) {
  Slot_t *s = &(g->b[i % NUMSLOTS]);
  s->q0 = q;
  for(int i = 0; i < NUMCHUNKS; i++)
    s->c[i] = c[i];
  s->q1 = q;
}
```

GRB Read Operation

```
grb_ret_t grb_read(Grb_t *g, size_t i, QID *q, Chunk c[NUMCHUNKS]) {
  Slot_t *s = &(g->b[i % NUMSLOTS]);
  *q = s->q1;
  for(int i = NUMCHUNKS - 1; i >= 0; i--)
    c[i] = s->c[i];
  return *q == s->q0 ? GRB_OK : E_GRB_FAIL;
}
```

In practice, the loops can be replaced by `memcpy` calls.

Validation with `spin`

Check the GRB correctness property using the model checker `spin` [Holzmann].

Assumptions baked into the `spin` model:

1. QIDs are “fresh”
2. atomic reads and writes of QIDs and Chunks
3. hardware respects program order
4. memory is SC (*sequentially consistent*)

Result(s): the property holds.

Reality vs. Models

Problem:

1. Compilers may like to reorder memory accesses.
2. Multi-core ARMv8 is not SC!

No surprise: testing our prototype GRBs on pilot HW reveals undetected read violations.
None where due to the compiler (some older `gcc`).

GRB Read Operation with Fences

```
grb_ret_t grb_read(Grb_t *g, size_t i, QID *q, Chunk c[NUMCHUNKS]) {
  Slot_t *s = &(g->b[i % NUMSLOTS]);  /* safety mod */
  *q = s->q1;
  PSO_lfence();
  for(int i = NUMCHUNKS - 1; i >= 0; i--)
    c[i] = s->c[i];
  PSO_lfence();
  return *q == s->q0 ? GRB_OK : E_GRB_FAIL;
}
```

GRB Write Operation with Fences

```
void grb_write(Grb_t *g, size_t i, QID q, Chunk c[NUMCHUNKS]) {
  Slot_t *s = &(g->b[i % NUMSLOTS]);
  s->q0 = q;
  sfence();
  for(int i = 0; i < NUMCHUNKS; i++)
    s->c[i] = c[i];
  sfence();
  s->q1 = q;
}
```

Adding a third `sfence();` at the end actually reduces the likelihood of failed reads.

Fence Implementation for AArch64

```
inline void sfence(void) {
  asm ("DSB ISHST": : :"memory");
}

inline void PSO_lfence(void) {
  asm ("DSB ISHLD": : :"memory");
}
```

Back to `spin`

There are generic memory models in the literature, e.g., by Matsumoto et al. [2018] based on previous work by the same group that probably started with Abe and Maeda [2014].

Result(s):

- Modelling weak memory is expensive (in terms of state space sizes).
- Fences are necessary. Even for the PSO model, just two fences in the `grb_write` are enough.

Back to `spin`

There are generic memory models in the literature, e.g., by Matsumoto et al. [2018] based on previous work by the same group that probably started with Abe and Maeda [2014].

Result(s):

- Modelling weak memory is expensive (in terms of state space sizes).
- Fences are necessary. Even for the PSO model, just two fences in the `grb_write` are enough.
- Conclusion: this PSO model isn't weak enough! It doesn't consider reordering of reads.

Is This a Real Issue?

Problem: Could we reproduce any undetected read violations on the pilot when the writer had its fences?

Answer: Not reliably, even, after hours of hammering pilot hardware.

Problem: There are many different variations of fence instructions on these ARM chips. What's correct? What's best?

How to Improve Testing Fence Arrangements

Use the `diy` tool suite [diy, 2021] (nowadays called `herdtools7`) to encode the reader and writer core logic with varying fence arrangements.

Evaluate by running `diy`-generated binaries on the pilot, and randomise timing and affinities to find correctness property violations.

A diy Model

```
AArch64 grb-arm-WdmbishldRdmbish
{
0: X1=q0; 0: X2=c; 0: X3=q1;
1: X1=q0; 1: X2=c; 1: X3=q1;
1: X4=p0; 1: X5=d; 1: X6=p1;
}
  P0             | P1          ;
  MOV X0,#1      | LDR X0,[X3];
  STR X0,[X1]    | STR X0,[X6];
  DMB ISHLD      | DSB ISH     ;
  MOV X0,#2      | LDR X0,[X2];
  STR X0,[X2]    | STR X0,[X5];
  DMB ISHLD      | DSB ISH     ;
  MOV X0,#1      | LDR X0,[X1];
  STR X0,[X3]    | STR X0,[X4];
exists
(p0=1 /\ d=0 /\ p1=1)
```

Running this 10^{10} times on a pilot took less than an hour and resulted in

```
Histogram (8 states)
4255112670:>[d]=0; [p0]=0; [p1]=0;
45495453:>[d]=2; [p0]=0; [p1]=0;
15276213:>[d]=0; [p0]=1; [p1]=0;
25947560:>[d]=2; [p0]=1; [p1]=0;
306118224:>[d]=0; [p0]=0; [p1]=1;
687405161:>[d]=2; [p0]=0; [p1]=1;
19486279*>[d]=0; [p0]=1; [p1]=1;
4645158440:>[d]=2; [p0]=1; [p1]=1;
Ok

Witnesses
Positive: 19486279, Negative: 9980513721
Condition exists ([p0]=1 /\ [d]=0 /\ [p1]=1) is validated
```

Analysis

This particularly stupid fence arrangement has a non-zero (about 0.19%) probability of incorrectness.
Using `DMB SY` on the writer side and *no* fence on the reader side performed better, with only 29 incorrect behaviours in 10^{10}.
Then using `DSB SY` or similar on the reader side gave 0 incorrect behaviours even over much longer test periods.

Result(s): Read fences are necessary. Some fence arrangements are *almost* reliable with error probabilities below 10^{-9}. We would have a hard time finding these bugs with our previous testing regime.

The Endgame in GRB Verification

Why don’t we just verify it?
The current SOTA in verification of concurrent, racy programs with fences on weak memory multi-core HW is a research problem.
There’s initial work by Mansky et al. [2017] to beef up IRIS/VST to problems like this, but it’s not done yet.

The Endgame in GRB Verification

Why don't we just verify it?
The current SOTA in verification of concurrent, racy programs with fences on weak memory multi-core HW is a research problem.
There's initial work by Mansky et al. [2017] to beef up IRIS/VST to problems like this, but it's not done yet.

Talk to me if you think we're doing it wrong or not using the right tools!

References I

diy. https://github.com/herd/herdtools7, 2021. Last accessed 2021/10/21.

Tatsuya Abe and Toshiyuki Maeda. A general model checking framework for various memory consistency models. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 332–341, 2014. URL https://doi.org/10.1109/IPDPSW.2014.47.

Gerard Holzmann. Spin. http://spinroot.com/spin/whatispin.html. Accessed 2019/10/21.

William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. *Proc. ACM Program. Lang.*, 1 (OOPSLA), October 2017. URL https://doi.org/10.1145/3133911.

References II

Kosuke Matsumoto, Tomoharu Ugawa, and Tatsuya Abe. Improvement of a library for model checking under weakly ordered memory model with SPIN. *Journal of Information Processing*, 26:314–326, 2018. URL https://doi.org/10.2197/ipsjjip.26.314.

Glossary

GRB	*glorified ring buffer*, a wait-free data structure (🫆)
GPS	*global positioning system*, a satellite-based radionavigation system
IMU	*inertial movement unit*, a motion sensor
QID	*quamtum ID*, a nonce-like entity