tips and tricks to tune your software for arm

Krunal Bauskar
(driving #dbonarm initiative)

quick word about me

- working in mysql space for more than decade now.
- currently working @huawei driving the #dbonarm initiative with aim to make open source database ecosystem available on arm in optimal fashion.
- in past worked with
  - percona (as percona xtradb cluster product lead)
  - oracle/mysql (innodb developer)
  - yahoo! labs (big-data research engineer)
  - teradata (mysql fastest storage engine)
  - and more...
- blog: https://mysqlonarm.github.io/
- present at all mysql leading conferences: fosdem, percona live, mariadb fest, osi, minerva db, ….

HUAWEI

PERCONA

MySQL

YAHOO! LABS

Persistent

teradata.

Informatica

IBM

cisco

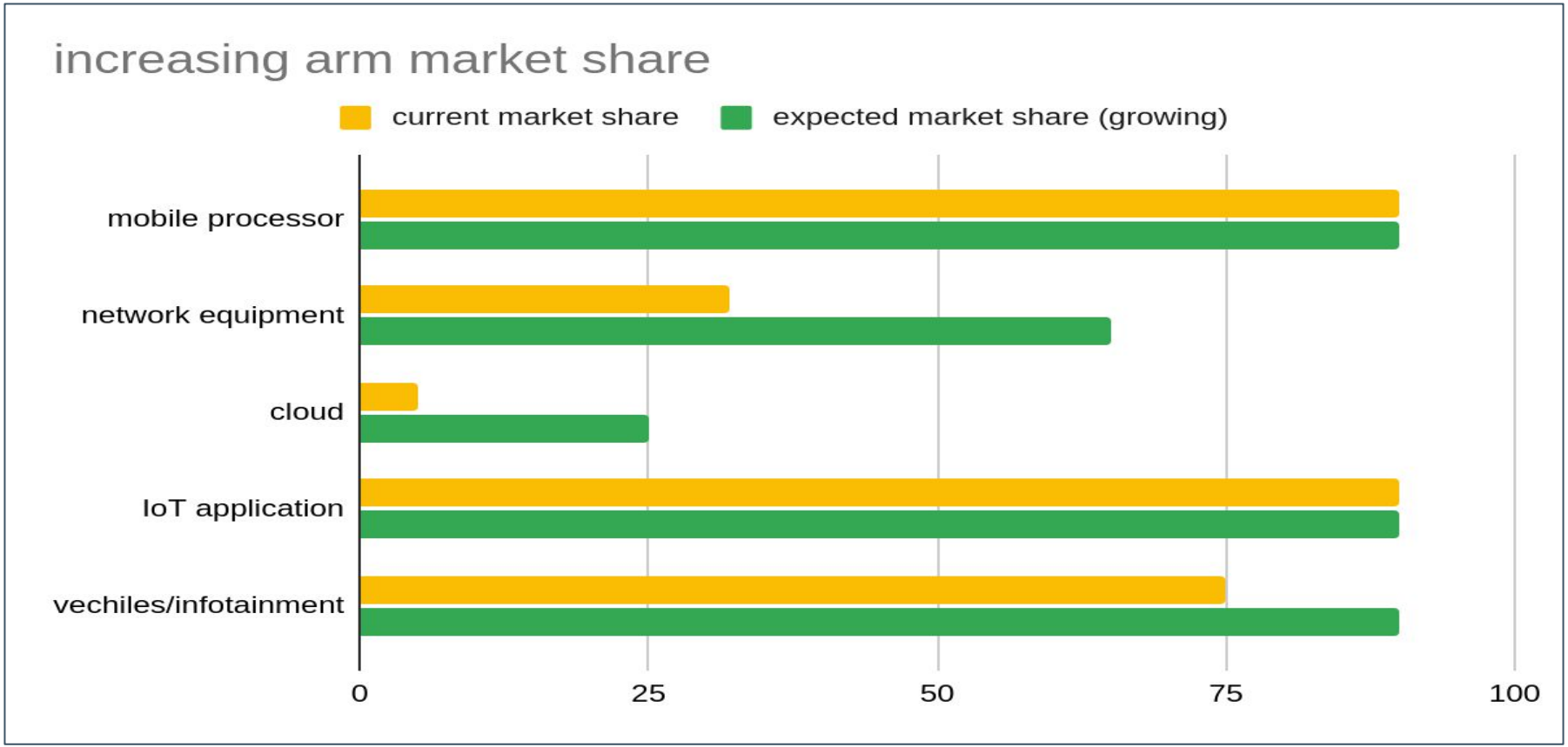
agenda

- growing popularity of arm
- why tune for arm?
- tried and tested tips for tuning your software on arm
- future work

agenda

- growing popularity of arm
- why tune for arm?
- tried and tested tips for tuning your software on arm
- future work

growing popularity of the arm



https://www.statista.com/statistics/1132112/arm-market-share-targets/

growing popularity of the arm

- growing vendor offering arm cloud-instances/processor:
  - huawei - kunpeng-920
  - amazon - graviton
  - oracle - ampere
  - apple - m1
  - microsoft/google*
- in 2020, 49% of the new instances booted on aws were arm powered.
- all major/popular software/projects are now available on arm
  - mysql, mariadb, cassandra, mongodb, redis, clickhouse, columnstore, etc..
  - big data/hadoop framework
  - application servers: nginx
  - tools, libraries

thanks to price-performance and more cores that helps get more things done in parallel

growing popularity of the arm

growing market share

growing developer base

growing cloud adoption

growing user interest

is your software ready for next big thing?

agenda

- growing popularity of arm
- why tune for arm?
- tried and tested tips for tuning your software on arm
- future work

why tune for arm?

“arm is little endian so my software will work on arm”

yes this is true but there is difference between

“works for arm”

vs

“tuned for arm”

why tune for arm?

- getting software to work on arm?
  - majority of the software will work out-of-box on arm with least efforts. (needs compiling. no binary level compatibility).
  - 3rd party dependencies are getting resolved with all major libraries being made available on arm.
  - software are using iterative approach allowing them to get core functionalities available on arm and then expanding the additional offering.

…. easy step is done.

why tune for arm?

- arm is different.
  - instruction set especially when code uses them directly. Say crc32,sha, etc..
  - memory model (relaxed)
  - more cores in turns more numa nodes. (numa scalability challenges).
  - atomic operation difference/optimization (lse).
  - simd/acle instructions for parallel processing
  - timer register, control register
  - processor/core affinity (no hyperthreading and turbo-mode)

… all this make it necessary to profile and study a software on arm and tune it for optimal performance.

agenda

- growing popularity of arm
- why tune for arm?
- tried and tested tips for tuning your software on arm
- future work

#1 - reassess memory model and ensure optimal memory barrier/order is used

#1 - memory order

- arm has weaker memory model. majority of the softwares were tuned for stronger memory model.
- softwares are moving to atomics/lock-free programming.

global counters often uses atomics but with default memory order **memory_order_seq_cst.**

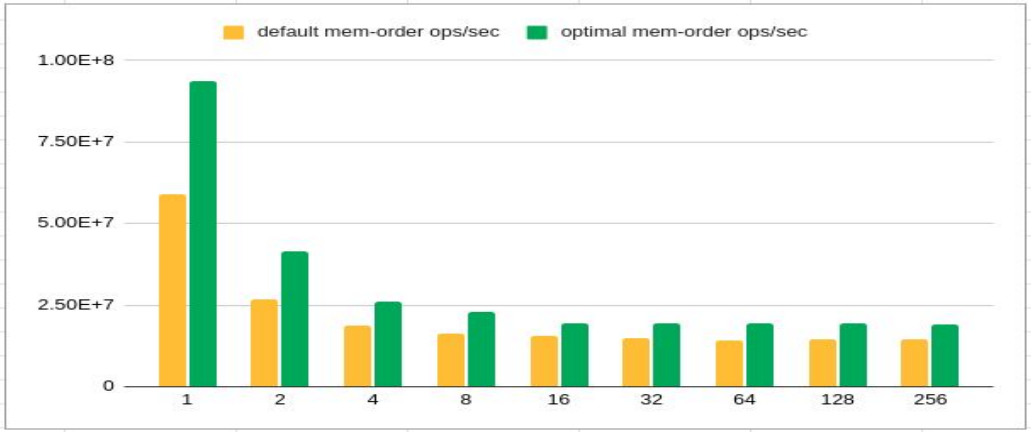
given they are not meant for synchronization, memory order could be replaced with **memory_order_relaxed.**

spin-loop/coordinating atomic variables too often uses default memory order **memory_order_seq_cst.**

switching to use **release/acquire memory** barrier/order should help.

** mutex misuse: look-out if you really need mutex (mutual exclusion) or lock (reader/writer protocol) could do.*

#1 - memory order



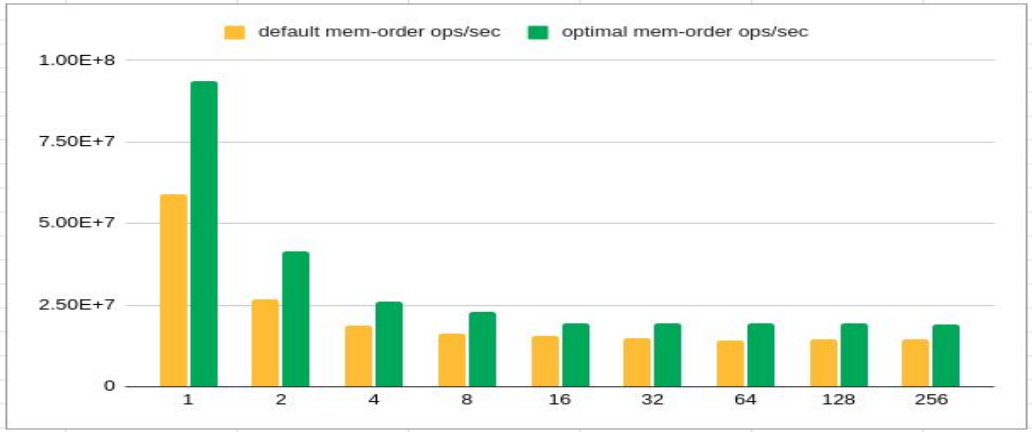
operation per seconds is better with relaxed memory order

```
unsigned long request;

request++;
```

```
atomic<unsigned long> request;

request++;
(continue to use default memory order)
```

often this tend to get missed given simple increment operator. user tend to look out for known atomic functions.

#1 - memory order



operation per seconds is better with relaxed memory order

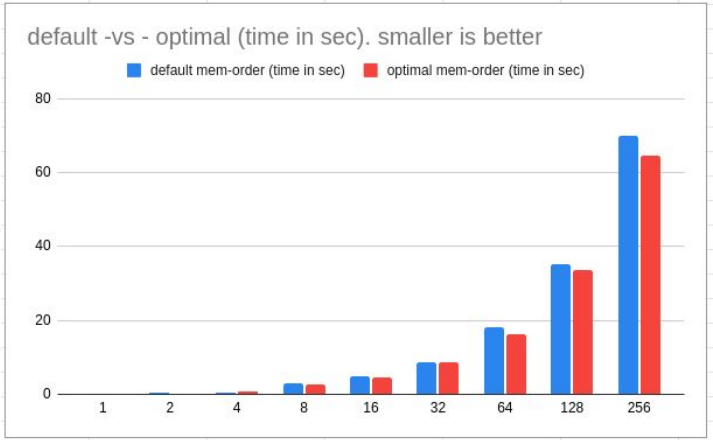
```
unsigned long request;

request++;
```

```
atomic<unsigned long> request;

request++;
(continue to use default memory order)
```

```
atomic<unsigned long> request;

request.fetch_add(1,
std::memory_order_relaxed)
```

#1 - memory order

```
atomic<bool> lock{false};

void lock_func() {
  bool expected = false;
  lock.compare_exchange_strong(expected, true);
}
void unlock_func() {
  lock.store(false);
}
```




```
atomic<bool> lock{false};

void lock_func() {
  bool expected = false;
  lock.compare_exchange_strong(expected, true, std::memory_order_acquire);
}
void unlock_func() {
  lock.store(false, std::memory_order_release);
}
```

also, explicitly memory order helps clear the intention of what is expected from the said flow.

#1 - memory order

most important aspect to look out for.

with mysql, 50% of the issues reported were around usage of optimal memory order.

majority of the issues helped scale mysql in range of 3-5%.

#2 - low-level functions

#2 - low level functions

- often software has assembly code for some low level functions.
  - virtual timer (random number generator)
    ```
    __asm __volatile("mrs       %0, CNTVCT_EL0" : "=&r" (result));
    ```
  - memory barrier: `dmb, dsb, isb, etc…`
    ```
    __asm__ __volatile__("isb" ::: "memory")
    ```
  - pause/yield (very much needed during spin loop)
    ```
    __asm__ __volatile__("yield");
    ```
  - accessing control register to find out processor capabilities

..trick I follow is scan for all _x86_64 and add relevant aarch64 specific code.

#2 - low level functions

.. follow-up fix may be needed since the aarch64 instruction may not generate the said effect.

- yield is not direct replacement to x86 PAUSE.
- timer for clock cycle could be different.
- memory barrier introduces additional latency.
- tuning logic based on processor clock frequency.

it is difficult:
with mysql/mariadb, we still continue to tune pause logic on arm for spin-loop.

#3 - programming for more numa nodes

#3 - more numa nodes

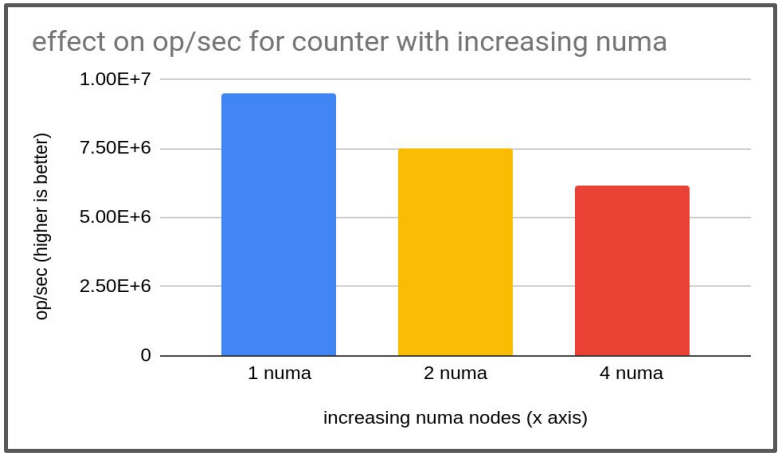
- arm processor offer lot of cores in turn more numa node.
- this introduces its own set of challenges: cross numa movement, numa latency, numa thread affinity, etc...

#3 - more numa nodes

- arm processor offer lot of cores in turn more numa node.
- this introduces its own set of challenges: cross numa movement, numa latency, numa thread affinity, etc...

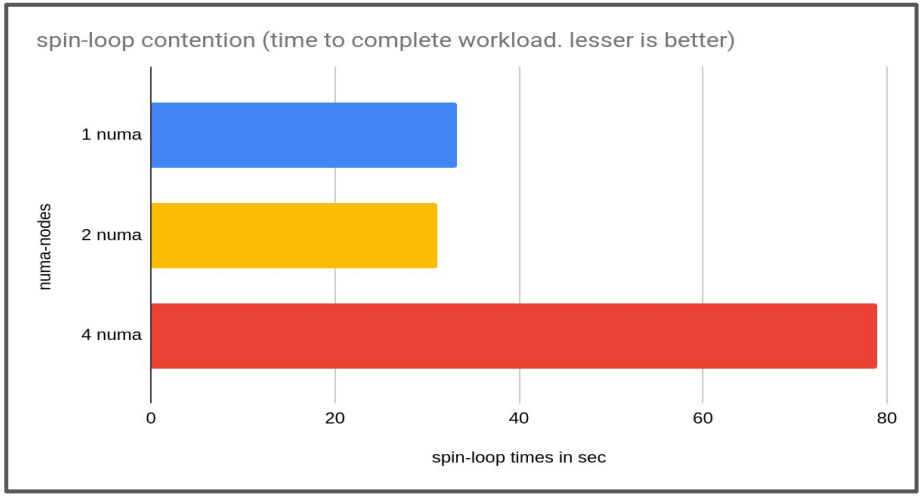
traditionally, when software hits IO/CPU contention then besides making optimal use of the resources other way to help ease the contention is increase the resources.

#3 - more numa nodes

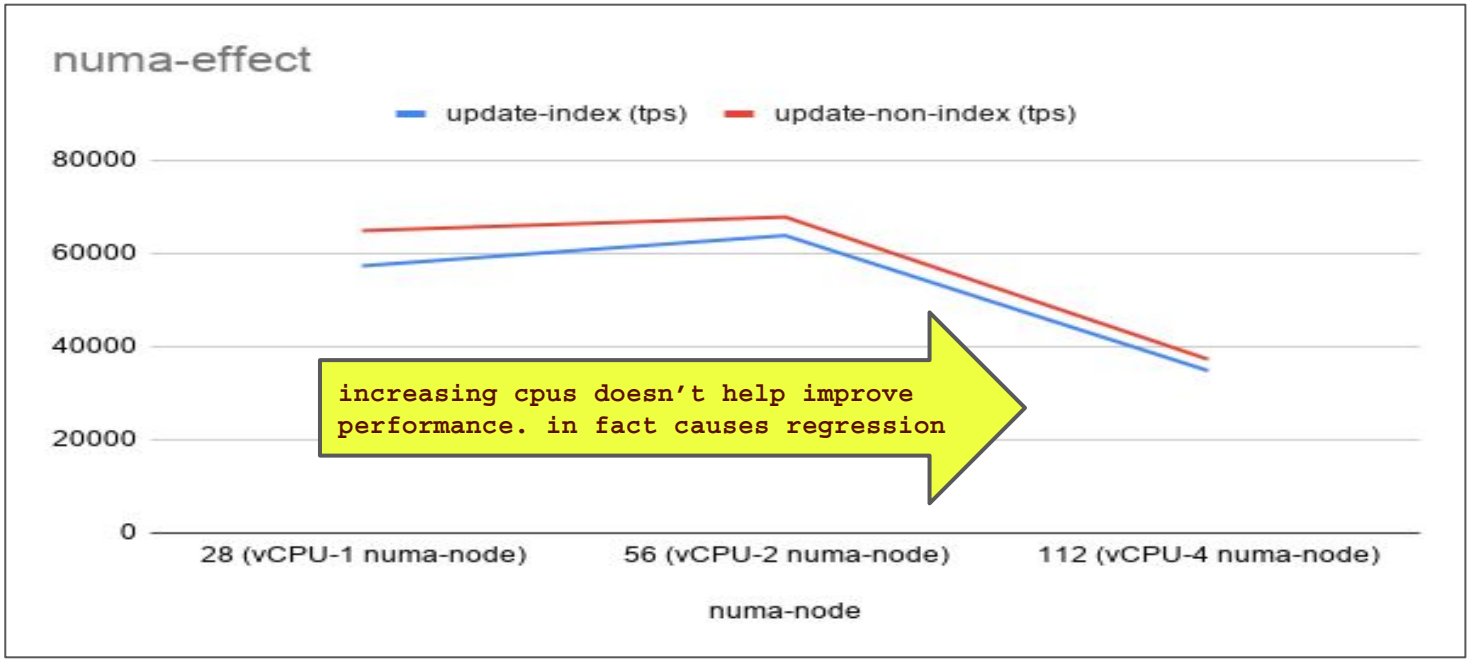


with increasing numa-nodes. ops/sec to increment counter is reducing.

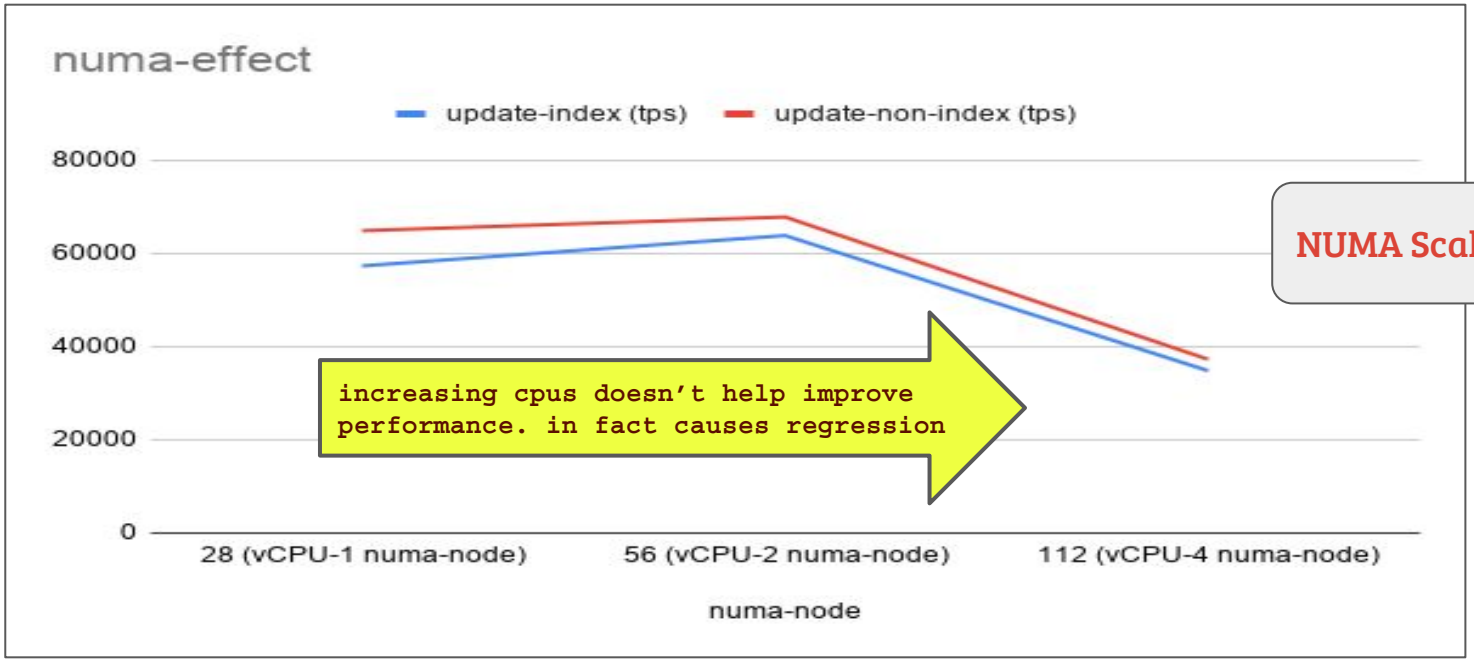
With increasing numa, spin loop workload is taking more time despite of having more CPU cores so ideally lesser contention.



#3 - more numa nodes



#3 - more numa nodes

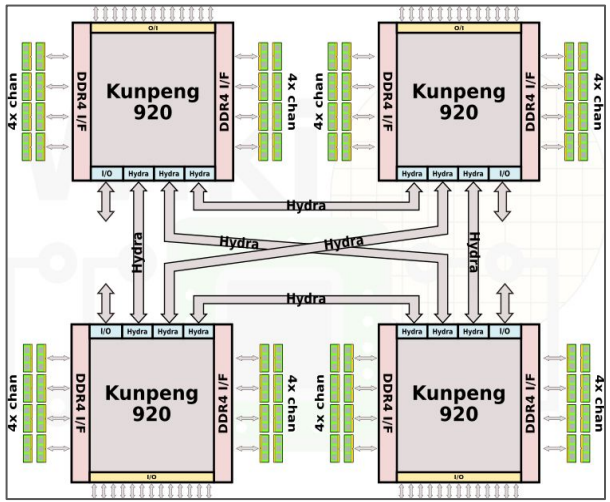


NUMA Scalability

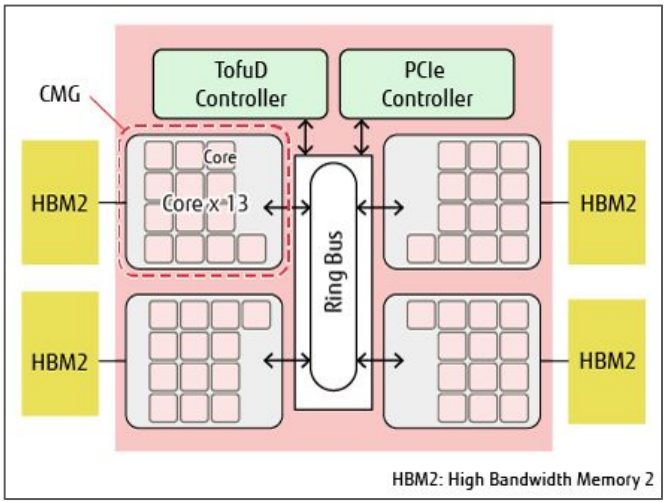
#3 - more numa nodes

arm processor generally tend to have more cores but may be less powerful.

also, numa is no more per socket. single cpu socket could have 2/4 numa nodes.



Kunpeng 920 4P



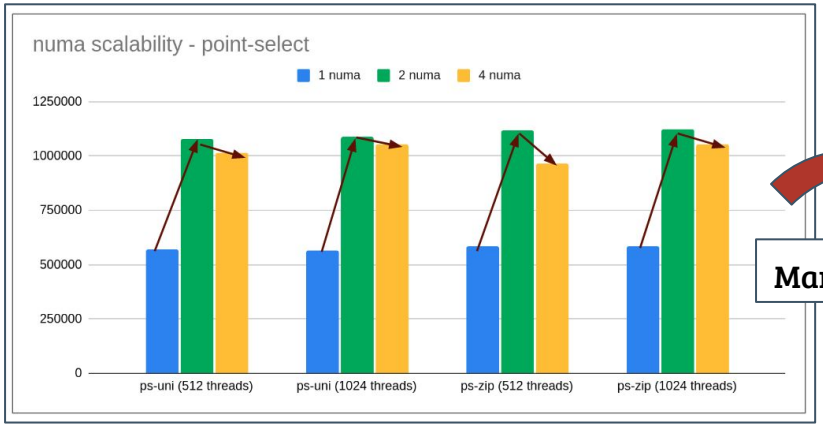
A64FX

#3 - more numa nodes

how to handle NUMA bottleneck?

- even distribution of memory across all numa nodes.
  `(interleave=all/set_mempolicy(MPOL_INTERLEAVE)`
- localized numa processing (workload and data located on same numa node).
- lesser cross-numa movement of the threads. setting CPU affinity for the threads.
- distributed counter, locks, global objects (hashmap/list/etc.…).
- even workload distribution. ensuring all threads are working on all numa nodes.
- non-shared architecture.

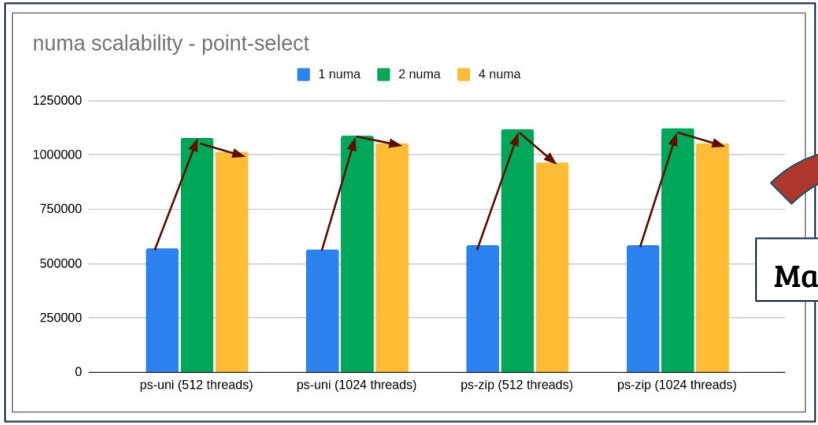
#3 - more numa nodes



MariaDB-10.6

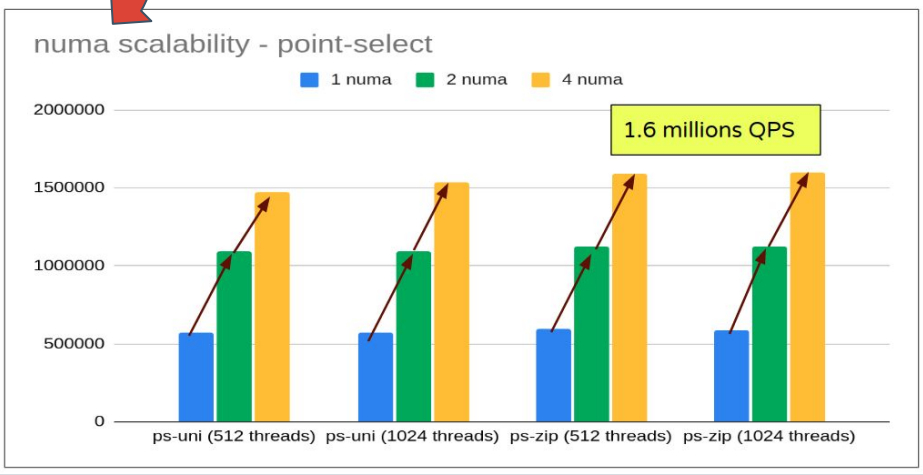
Fixed numa bottleneck
global mutex -> object/table mutex
global counter -> distributed counter

#3 - more numa nodes



MariaDB-10.6

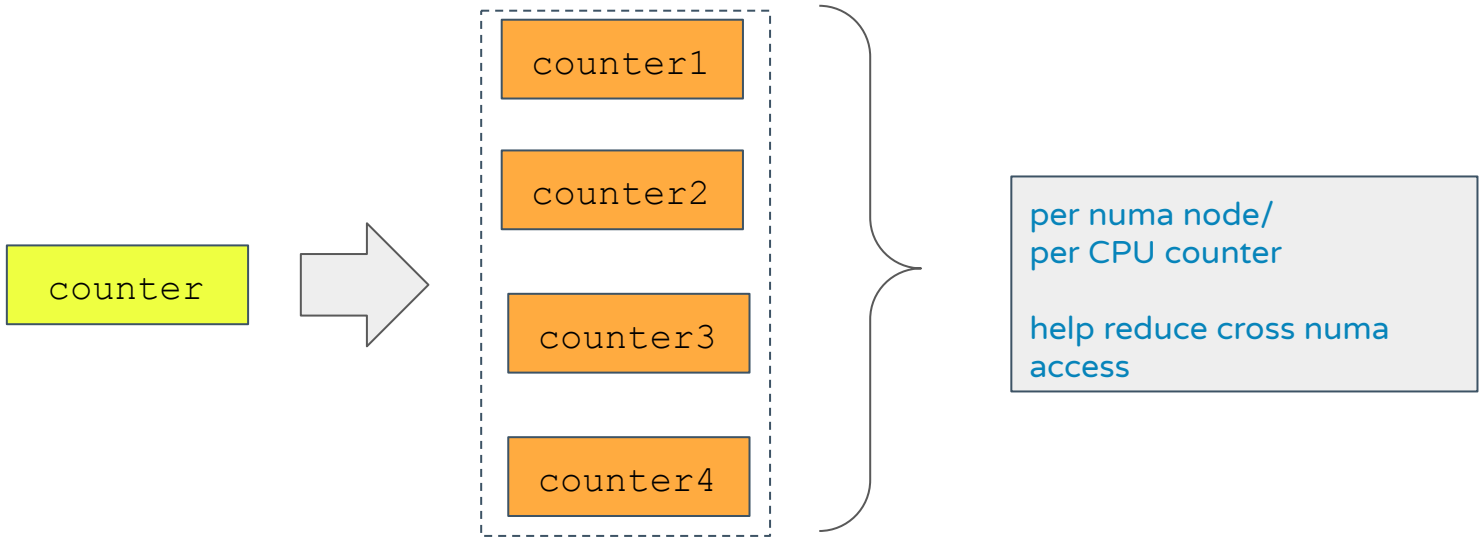
Fixed numa bottleneck
global mutex -> object/table mutex
global counter -> distributed counter



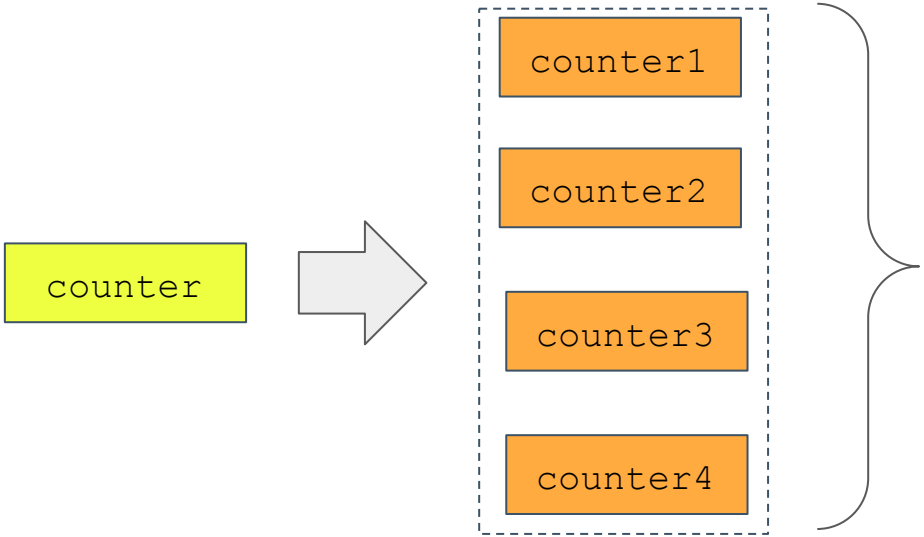
removing smaller contention could have wider impact in NUMA scale.

wait-graph is quite different with NUMA.

#3 - more numa nodes



#3 - more numa nodes



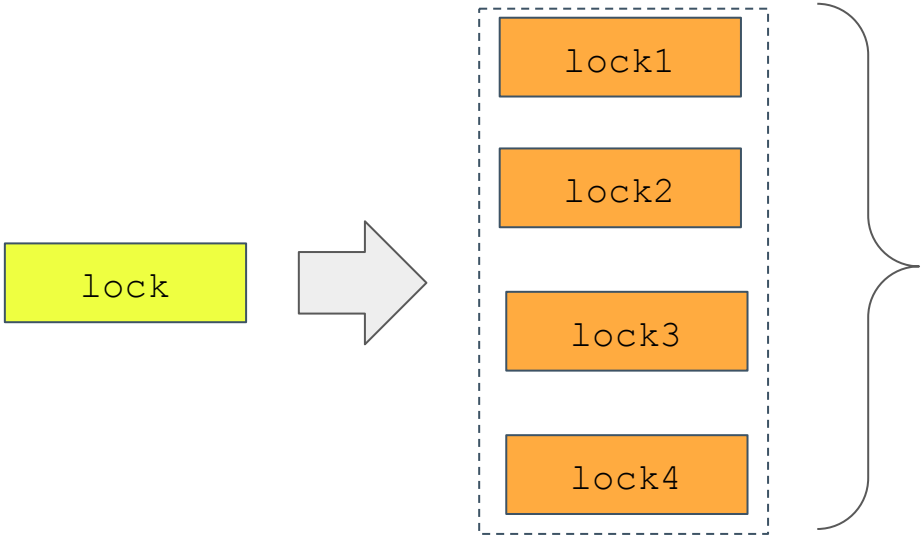
CHALLENGES

Let's say we place a counter per CPU/core.

Now getting the core-id with each increment using sched_getcpu is costly in arm.

(sched_getcpu is not implemented as vdso like it is done in x86_64)

#3 - more numa nodes



each lock represent set of resources.

read-lock could be obtained by pinning respective lock

write-lock needs pinning all locks.

On same line, one can explore scalable **distributed locks**.

#4 - mind you cache line

#4 - cache line difference

- arm has wider cache line. 128 bytes or mix of 64/128 (L1/L2/L3).
- global state variables, counters, structures should be cache line aligned to avoid false sharing.
- often as programmer we hit this dilemma. overdoing leads to memory bloat.

ensure the padding or cache line alignment is tuned to consider arm.

- → global counters
- → global state variables
- → structure or specific element
- → locks
- → mutexes

- ➢ padding vs alignas (C++)
- ➢ re-organizing the structure to host hot-variable and cold-variable(s) on same cacheline

#5 - branching/pipeline

#5 - branching/pipeline

- branching probably is most common/frequently done operation (after move).
- optimal branching could help make a visible difference in performance of software.
- how arm processor pipeline these operations, execute branching could be different from other processor (even other variant of arm)
- hint could be passed to the certain critical branching conditions.

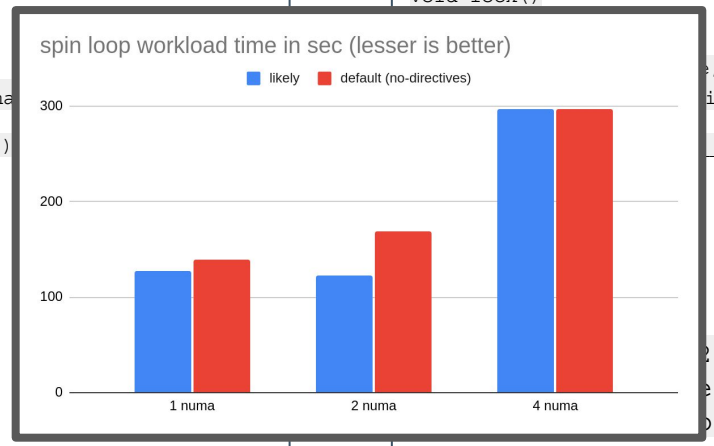
```
void lock()
{
  while (true) {
    bool expected = false;
    if (unlikely(!lock_unit.compare_exchange_strong(expected,
true))) {
      __asm__ __volatile__("" ::: "memory");
      continue;
    }
    break;
  }
}

unlikely()
Elapsed time: 40.1403 s
 86.09 |48:┌──b.eq    54
  9.27 |   |  strb    w4, [x5]
```

```
void lock()
{
  while (true) {
    bool expected = false;
    if (likely(!lock_unit.compare_exchange_strong(expected,
true))) {
      __asm__ __volatile__("" ::: "memory");
      continue;
    }
    break;
  }
}

likely()
Elapsed time: 38.2878 s
 83.58 |4c:┌──b.ne    6c
  8.90 |6c:└─→strb    w2, [x4]
```

#5 - branching/pipeline

- branching probably is most common/frequently done operation (after move).
- optimal branching could help make a visible difference in performance of software.
- how arm processor pipeline these operations, execute branching could be different from other processor (even other variant of arm)
- hint could be passed to the certain critical branching conditions.

```
void lock()
{
  while (true) {
    bool expected = false;
    if (unlikely(!lock_unit.compare_excha
true))) {
      __asm__ __volatile__("" ::: "memory")
      continue;
    }
    break;
  }
}

unlikely()
Elapsed time: 40.1403 s
 86.09 |48:┌──b.eq   54
  9.27 |   |  strb   w4, [x5]
```




```
void lock()
                                    ;
                                    it.compare_exchange_strong(expected,

                                    __("" ::: "memory");

                                    2878 s
                                    e    6c
                                    b    w2, [x4]
```

#5 - 64 bit processing

#5 - 64 bits operations

- Given software continue to support 32/64 bits there is a special optimization added for 64 bits with processor directives.

```
+#if defined(__x86_64__) || defined(__aarch64__)
+  for (; length >= 8; length -= 8, from += 8, to += 8) {
+    if (uint8korr(from) & 0x8080808080808080) break;
+    int8store(to, uint8korr(from));
+  }
+#endif /* defined(__x86_64__) || defined(__aarch64__) */
+
```

- Processing array in chunk of 64 bits is common optimization.
- Ensure that all such optimization are turned on for __aarch64__ too.

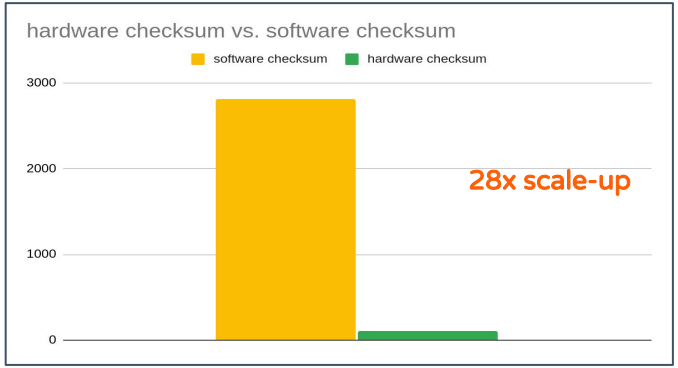
#6 - hardware level functions

#6 - hardware instructions

- Software uses hardware function to accelerate certain jobs.

C code: `crc = __crc32w(crc, *buf4);`

Translate too: `f54: 1ac44842 crc32w w2, w2, w4`



- For example:
  - Cryptographic: SHA, AES,
  - Random Number Generation
  - Direct Rounding
  - Floating point handling
  - Memory barriers are related like data barriers, instruction barriers, etc…
  - Some scheduling/interruption hints.
  - Swap of data.
  - Prefetch data/instruction memory.
  - ….And more data processing intrinsics.

#7 - enabling additional functionalities

#7 - atomics support (lse)

- arm supports LSE (large system extension) which helps reduce loop for atomic operation of load-store to a single instruction.
- given it is added functionality not all arm v8 processor support it.
- `-moutline-atomic` checks during the execution if processor can support LSE and accordingly execute the code. Unfortunately, this approach tend to incur some overhead as check happens for each evaluation.
- also, check if enabling lse for your software environment is really helpful.

#7 - atomics support (lse)

`-O2 -march=armv8-a -mno-outline-atomics`

```
.L5:
        ldaxrb  w3, [x0]
        cmp     w3, w1, uxtb
        bne     .L6
        stxrb   w4, w2, [x0]
        cbnz    w4, .L5
```

loop to single stmt

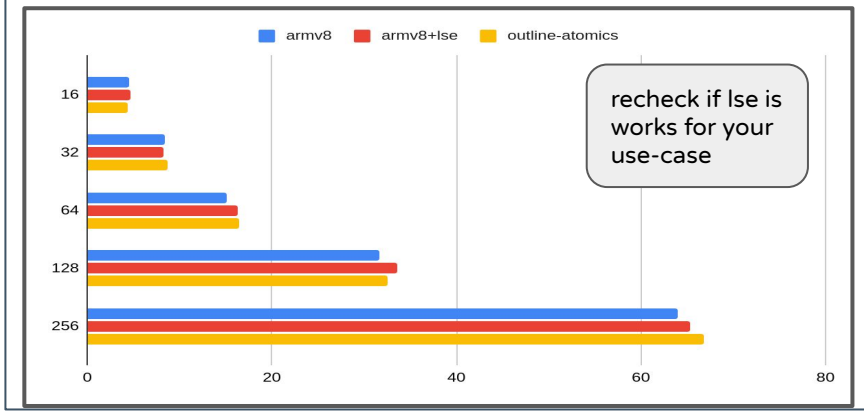
`-O2 -march=armv8-a+lse -mno-outline-atomics`

```
casab   w1, w2, [x0]
```

Single stmt to dynamic routine

`-O2 -march=armv8-a -moutline-atomics`

```
bl      __aarch64_cas1_acq
```



code	asm (perf output)

bool expected = true;
flag.compare_exchange_strong(expected, false);
	<__aarch64_cas1_acq_rel>:
	__aarch64_cas1_acq_rel():
	adrp x16, 11000 <__data_start>
	ldrb w16, [x16, #25]
	↓ cbz w16, 14
	casalb w0, w1, [x2]
	← ret
	14: uxtb w16, w0
	18: ldaxrb w0, [x2]
	cmp w0, w16
	↓ b.ne 2c
	stlxrb w17, w1, [x2]
	↑ cbnz w17, 18
	2c: ← ret

Added indirection shouldn't waive off the added gain from switching to use lse.

#8 - simd/neon instructions

#8 - smid/neon

- meant for parallel/vectorized operations.

under-utilized in general.

- use-case driven
  - crc32c computation using hardware is further parallelised using vectorized operations
  - often use-cases like mapping (in db world charset mapping) can exploit vectorized parallelism.
  - compression using neon instructions. (a test patch for page-compression in mysql was done using neon instruction that showed some promising result).
  - smid is best fit for column-driven databases.
  - smid used in multiplying 2 NUMERIC data-types in pgsql.
  - smid used to find out array size for binary representation of decimal.

auto-vectorization done by gcc (`-ftree-loop-vectorize`). Enabled with -O3 optimization.

agenda

- growing popularity of arm
- why tune for arm?
- tried and tested tips for tuning your software on arm
- future work

future work

- exploring/fine tuning more numa aspect.
- exploring how inter-socket communication link could be used to its fullest capacity (kunpeng: hydra interface).
- improving scale vector extension usage.

let’s remain connected

- mail:
  - krunalbauskar@gmail.com
  - mysqlonarm@gmail.com
- blog:
  - https://mysqlonarm.github.io/
- slack/community/forum channel:
  - #mysqlonarm
  - #mariadbonarm
- tweet: #mysqlonarm