Project #1: Concurrency and Contention Hotspot

This project teaches us how to detect a contention hotspot in a reader-writer lock and fix the contention using a distributed counter. There are three steps to implement a high-performance reader-writer lock in the DBMS:

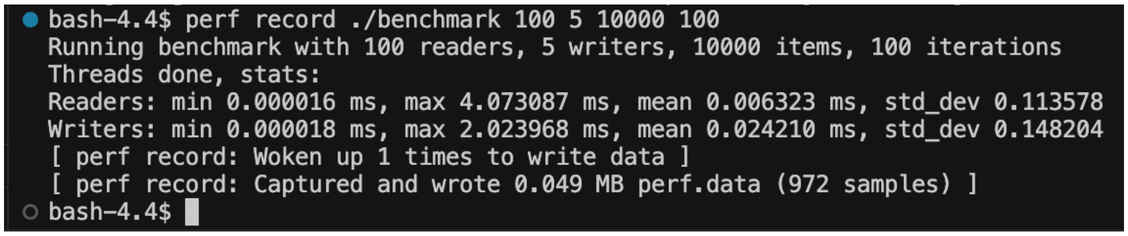
---

1. Profile the lock and identify the hotspot

In the initial phase of the project, our main objective was to profile the reader-writer lock and detect hotspots. We achieved this using the Linux tool perf, which helped us understand the bottleneck in the code and see the exact resources that require optimization.

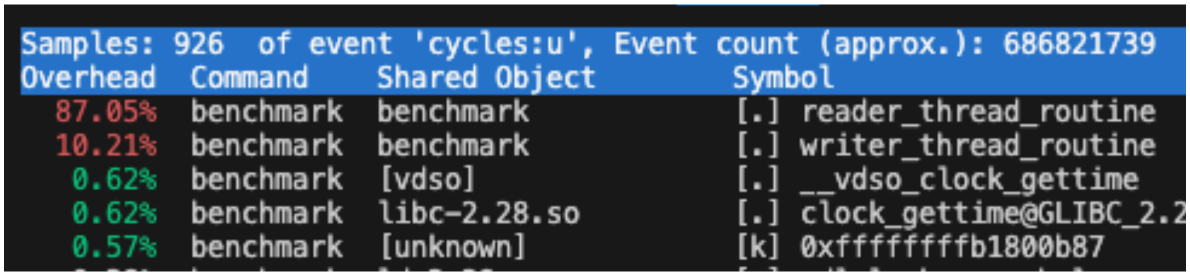
Steps:
- Changed the default terminal to bash on cade machine.
- `` `make benchmark` `` to build the code
- `` `perf record ./benchmark 100 5 10000 100` `` to run the benchmark and record the results. Here the arguments to the benchmark are, in order:
  1. nreaders: How many reader threads to launch
  2. nwriters: How many writer threads to launch
  3. nitems: # of items in the array
  4. niters: # of iterations. In each iteration a thread acquires the lock once

Output:



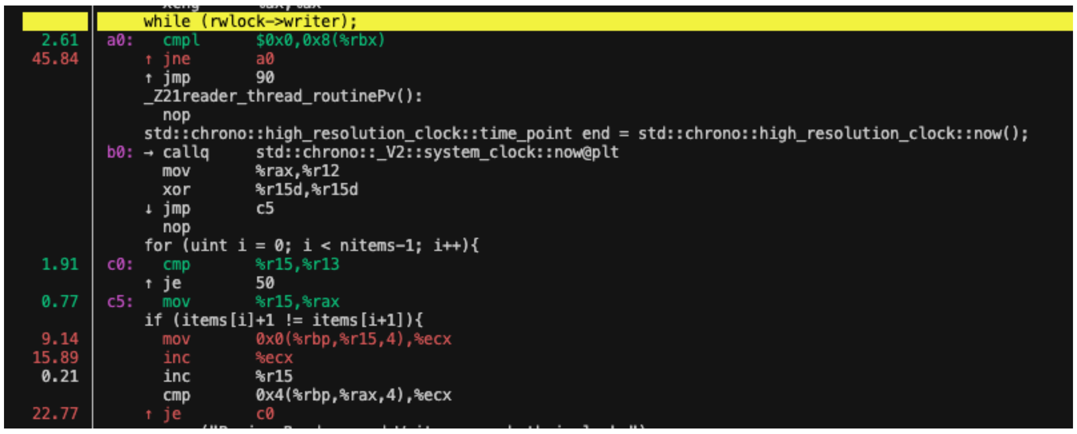
```
● bash-4.4$ perf record ./benchmark 100 5 10000 100
  Running benchmark with 100 readers, 5 writers, 10000 items, 100 iterations
  Threads done, stats:
  Readers: min 0.000016 ms, max 4.073087 ms, mean 0.006323 ms, std_dev 0.113578
  Writers: min 0.000018 ms, max 2.023968 ms, mean 0.024210 ms, std_dev 0.148204
  [ perf record: Woken up 1 times to write data ]
  [ perf record: Captured and wrote 0.049 MB perf.data (972 samples) ]
○ bash-4.4$ █
```

We get the benchmarking ticks set in the code and various stats related to it. Additionally, This generates a perf report that further helps us understand the contention that is happening.



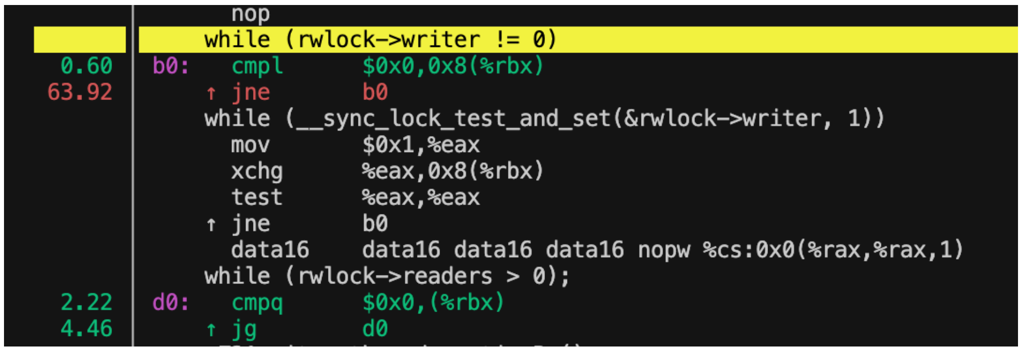
```
Samples: 926  of event 'cycles:u', Event count (approx.): 686821739
Overhead  Command    Shared Object         Symbol
  87.05%  benchmark  benchmark             [.] reader_thread_routine
  10.21%  benchmark  benchmark             [.] writer_thread_routine
   0.62%  benchmark  [vdso]                [.] __vdso_clock_gettime
   0.62%  benchmark  libc-2.28.so          [.] clock_gettime@GLIBC_2.2
   0.57%  benchmark  [unknown]             [k] 0xffffffffb1800b87
```

The two routines with the highest time spent (hotspots) are the reader and writer thread functions. Since we have a read heavy workload (100 reads and 5 writes), The max overhead is in the reader routine (87.05%).



```
                  while (rwlock->writer);
  2.61 │ a0:   cmpl      $0x0,0x8(%rbx)
 45.84 │     ↑ jne       a0
       │     ↑ jmp       90
       │     _Z21reader_thread_routinePv():
       │       nop
       │     std::chrono::high_resolution_clock::time_point end = std::chrono::high_resolution_clock::now();
       │ b0: → callq     std::chrono::_V2::system_clock::now@plt
       │       mov       %rax,%r12
       │       xor       %r15d,%r15d
       │     ↓ jmp       c5
       │       nop
       │     for (uint i = 0; i < nitems-1; i++){
  1.91 │ c0:   cmp       %r15,%r13
       │     ↑ je        50
  0.77 │ c5:   mov       %r15,%rax
       │     if (items[i]+1 != items[i+1]){
  9.14 │       mov       0x0(%rbp,%r15,4),%ecx
 15.89 │       inc       %ecx
  0.21 │       inc       %r15
       │       cmp       0x4(%rbp,%rax,4),%ecx
 22.77 │     ↑ je        c0
```

Reader_Thread_Routine Annotation: The red instructions are the hotspots of the routine and even if we were to ignore the hotspots in the benchmark.c code, we can observe that the maximum cpu% spent is due to the line in locks.c `` `while (rwlock->writer);` ``.



```
       │       nop
       │     while (rwlock->writer != 0)
  0.60 │ b0:   cmpl      $0x0,0x8(%rbx)
 63.92 │     ↑ jne       b0
       │     while (__sync_lock_test_and_set(&rwlock->writer, 1))
       │       mov       $0x1,%eax
       │       xchg      %eax,0x8(%rbx)
       │       test      %eax,%eax
       │     ↑ jne       b0
       │       data16    data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
       │     while (rwlock->readers > 0);
  2.22 │ d0:   cmpq      $0x0,(%rbx)
  4.46 │     ↑ jg        d0
```

Writer_Thread_Routine_Annotation: Hotspot is while waiting for other writers to finish.

Evaluation:

The line `` `while (rwlock->writer);` `` in the read_lock function experiences maximum overhead due to the nature of the workload as well (read-heavy) and the contention arising from shared resources protected by a lock. This behavior is noticeable from the perf report as well. In the above tested workload, multiple threads are primarily performing reads on shared resources, also providing exclusive access and priority to a write. The current read_lock implementation does not efficiently handle this scenario.

All the reader threads are contending to access the shared read variable. For instance, when a writer is detected, and multiple reader threads are concurrently executing, a critical contention point emerges. This contention arises from the fact that, for the writer to gain execution access, all the reader threads must individually decrement the same atomic variable. Consequently, this competition among the reader threads to access the same shared variable leads to contention and performance bottlenecks.

Furthermore, when a writer is detected, the reader threads enter a tightly spun while loop— `while (rwlock->writer);`. This particular loop consumes the highest proportion of execution time, accounting for approximately 45% of the total execution time. In a read-intensive workload, contention for the lock becomes particularly pronounced during the concurrent execution of this spinning loop. Each reader thread actively participates in this loop, continuously striving to determine the status of the writer. This intense competition for access to the `rwlock->writer` variable results in elevated CPU usage and a substantial overhead in contention, significantly affecting overall system performance.

---

5. Implement a distributed counter

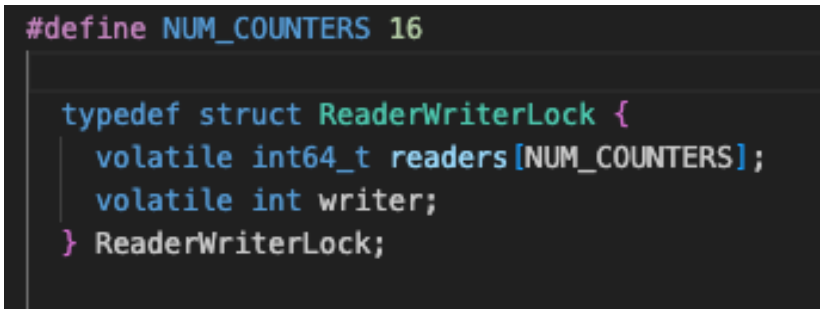
Now that we've identified the hotspot, To reduce the contention between the reader threads themselves we can make use of a distributed counter instead of a single reader variable.

Each thread has its own counter in this distributed array, which they increment when acquiring a read lock. This approach effectively distributes the tracking of readers across multiple counters, reducing contention when many reader threads are active concurrently. With the introduction of multiple reader counters, reader threads can update their respective counters without contending with each other. This fine-grained approach minimizes contention among readers, as each thread operates on its own counter without blocking or interfering with others.

The write lock still ensures writer priority and waits for all readers to drop their counters to zero. By introducing multiple reader counters, the modified code enables better parallelism for readers, reduces contention, and ensures that writers still have priority when they need access to the shared resource, particularly for read heavy workloads.

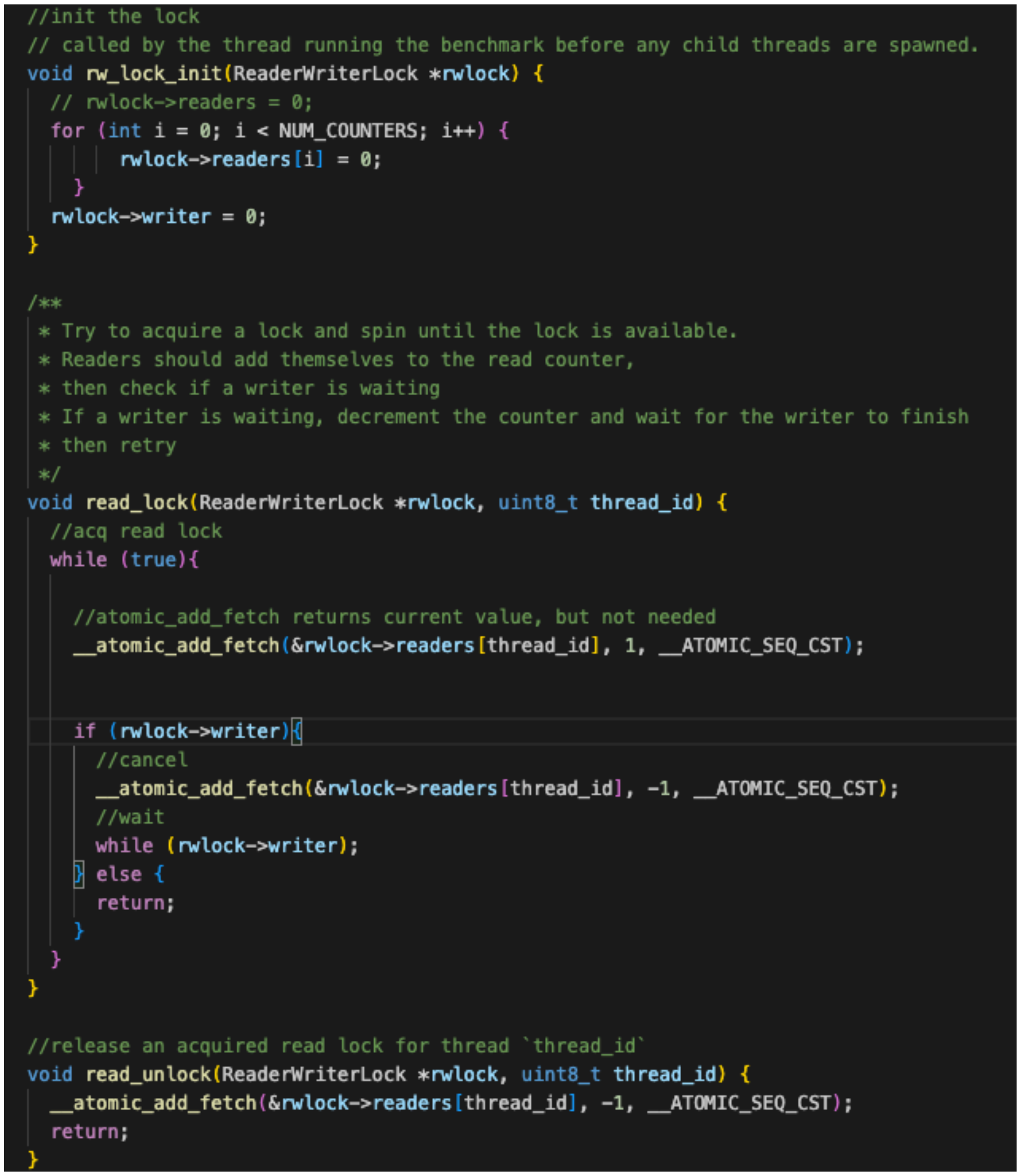
We set the array size as 16 on a 8 core machine (16 logical threads). With 16 counters, each thread has its own counter to increment or decrement. This approach allows each thread to work independently reducing contention from other threads when updating their respective counters. In a high-concurrency scenario, this minimizes contention and lock contention, leading to better parallelism. This also reduces false sharing between threads, where cache contention can also be an issue. By setting size to 16 we reduce chances of multiple threads trying to access the same cache line simultaneously.

Code changes:



```
#define NUM_COUNTERS 16

  typedef struct ReaderWriterLock {
    volatile int64_t readers[NUM_COUNTERS];
    volatile int writer;
  } ReaderWriterLock;
```

Changed reader counter to a distributed one

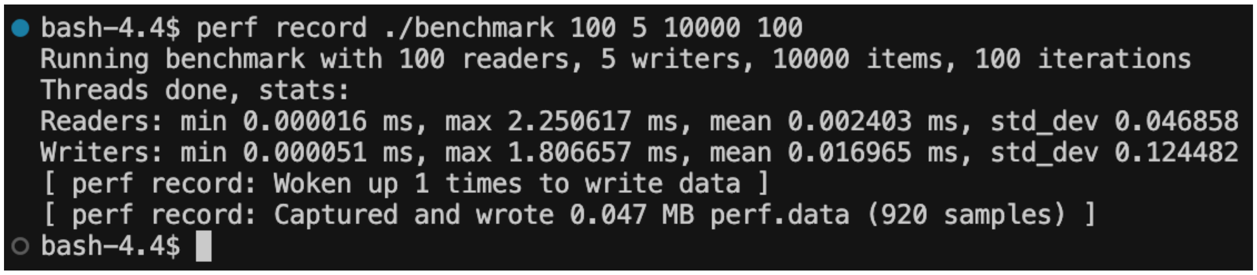


```
//init the lock
// called by the thread running the benchmark before any child threads are spawned.
void rw_lock_init(ReaderWriterLock *rwlock) {
  // rwlock->readers = 0;
  for (int i = 0; i < NUM_COUNTERS; i++) {
      rwlock->readers[i] = 0;
  }
  rwlock->writer = 0;
}

/**
 * Try to acquire a lock and spin until the lock is available.
 * Readers should add themselves to the read counter,
 * then check if a writer is waiting
 * If a writer is waiting, decrement the counter and wait for the writer to finish
 * then retry
 */
void read_lock(ReaderWriterLock *rwlock, uint8_t thread_id) {
  //acq read lock
  while (true){

    //atomic_add_fetch returns current value, but not needed
    __atomic_add_fetch(&rwlock->readers[thread_id], 1, __ATOMIC_SEQ_CST);


    if (rwlock->writer){
      //cancel
      __atomic_add_fetch(&rwlock->readers[thread_id], -1, __ATOMIC_SEQ_CST);
      //wait
      while (rwlock->writer);
    } else {
      return;
    }
  }
}

//release an acquired read lock for thread `thread_id`
void read_unlock(ReaderWriterLock *rwlock, uint8_t thread_id) {
  __atomic_add_fetch(&rwlock->readers[thread_id], -1, __ATOMIC_SEQ_CST);
  return;
}
```

Read lock changes (Refer to code for detailed changes)

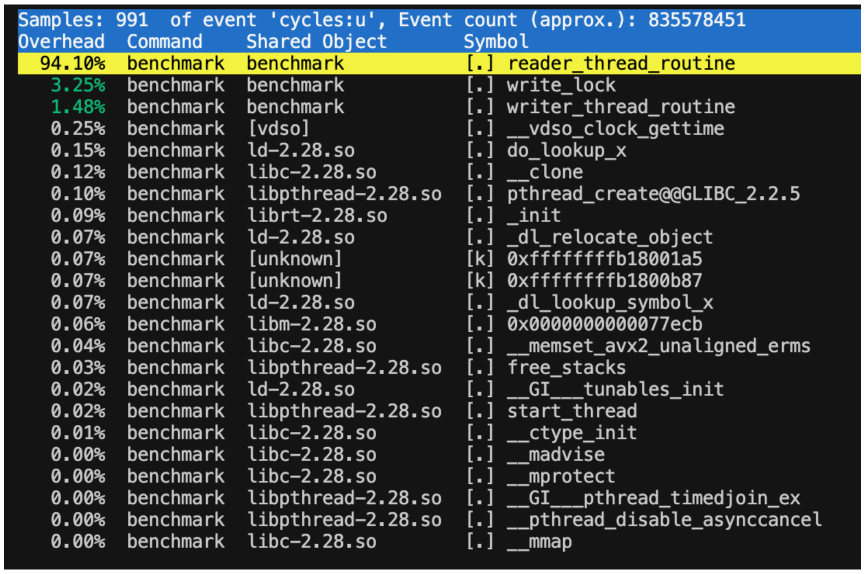
6. Test, Profile again, and Evaluate the performance

Finally after the changes, the benchmark was run again and below are the results:



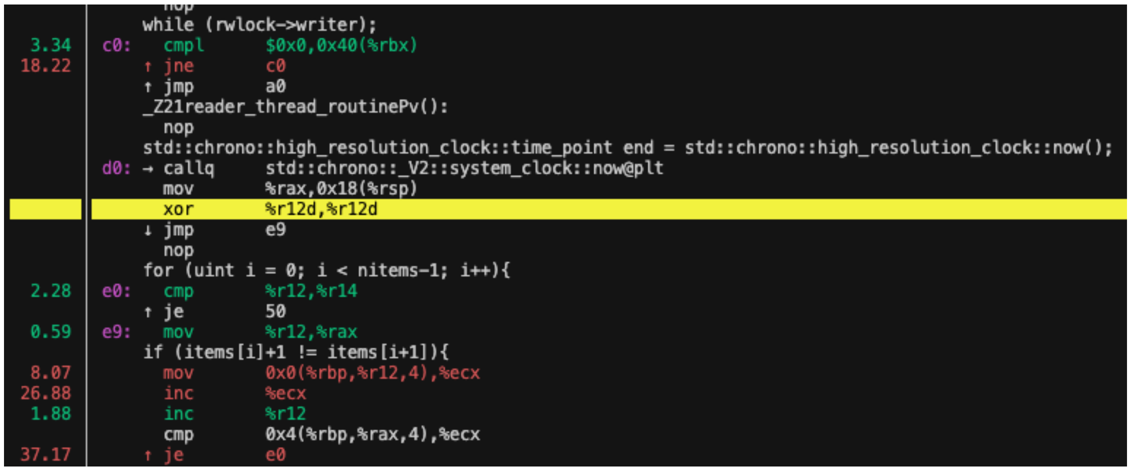
```
bash-4.4$ perf record ./benchmark 100 5 10000 100
Running benchmark with 100 readers, 5 writers, 10000 items, 100 iterations
Threads done, stats:
Readers: min 0.000016 ms, max 2.250617 ms, mean 0.002403 ms, std_dev 0.046858
Writers: min 0.000051 ms, max 1.806657 ms, mean 0.016965 ms, std_dev 0.124482
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.047 MB perf.data (920 samples) ]
bash-4.4$
```

The max (4.07 to 2.25 for read, 2.02 to 1.8 for write) and mean (0.006 to 0.002 for read, 0.024 to 0.016 for write) time reduces for both readers and writers



```
Samples: 991  of event 'cycles:u', Event count (approx.): 835578451
Overhead  Command    Shared Object        Symbol
  94.10%  benchmark  benchmark            [.] reader_thread_routine
   3.25%  benchmark  benchmark            [.] write_lock
   1.48%  benchmark  benchmark            [.] writer_thread_routine
   0.25%  benchmark  [vdso]               [.] __vdso_clock_gettime
   0.15%  benchmark  ld-2.28.so           [.] do_lookup_x
   0.12%  benchmark  libc-2.28.so         [.] __clone
   0.10%  benchmark  libpthread-2.28.so   [.] pthread_create@@GLIBC_2.2.5
   0.09%  benchmark  librt-2.28.so        [.] _init
   0.07%  benchmark  ld-2.28.so           [.] _dl_relocate_object
   0.07%  benchmark  [unknown]            [k] 0xffffffffb18001a5
   0.07%  benchmark  [unknown]            [k] 0xffffffffb1800b87
   0.07%  benchmark  ld-2.28.so           [.] _dl_lookup_symbol_x
   0.06%  benchmark  libm-2.28.so         [.] 0x0000000000077ecb
   0.04%  benchmark  libc-2.28.so         [.] __memset_avx2_unaligned_erms
   0.03%  benchmark  libpthread-2.28.so   [.] free_stacks
   0.02%  benchmark  ld-2.28.so           [.] __GI___tunables_init
   0.02%  benchmark  libpthread-2.28.so   [.] start_thread
   0.01%  benchmark  libc-2.28.so         [.] __ctype_init
   0.00%  benchmark  libc-2.28.so         [.] __madvise
   0.00%  benchmark  libc-2.28.so         [.] __mprotect
   0.00%  benchmark  libpthread-2.28.so   [.] __GI___pthread_timedjoin_ex
   0.00%  benchmark  libpthread-2.28.so   [.] __pthread_disable_asynccancel
   0.00%  benchmark  libc-2.28.so         [.] __mmap
```

New percentages after counter changes (writer cpu% decreases)



```
              nop
            while (rwlock->writer);
  3.34  c0:   cmpl      $0x0,0x40(%rbx)
 18.22        ↑ jne       c0
              ↑ jmp       a0
            _Z21reader_thread_routinePv():
              nop
            std::chrono::high_resolution_clock::time_point end = std::chrono::high_resolution_clock::now();
        d0: → callq     std::chrono::_V2::system_clock::now@plt
              mov       %rax,0x18(%rsp)
              xor       %r12d,%r12d
              ↓ jmp     e9
              nop
            for (uint i = 0; i < nitems-1; i++){
  2.28  e0:   cmp       %r12,%r14
              ↑ je      50
  0.59  e9:   mov       %r12,%rax
            if (items[i]+1 != items[i+1]){
  8.07        mov       0x0(%rbp,%r12,4),%ecx
 26.88        inc       %ecx
  1.88        inc       %r12
              cmp       0x4(%rbp,%rax,4),%ecx
 37.17      ↑ je        e0
```

Although reader routine overall percentage didn't reduce, The hotspot at `` `while (rwlock->writer);` `` has reduced to 18% from 45% and the new hotspot is in the benchmark.c code (validation code)

Performance improved at following Lines improved due to array counter implementation:

1) Independent counter decrement by threads once writer detected: `` `__atomic_add_fetch(&rwlock->readers[thread_id], -1, __ATOMIC_SEQ_CST);` ``

2) Less wait and time spent spinning: `` `while (rwlock->writer);` ``

Evaluation:

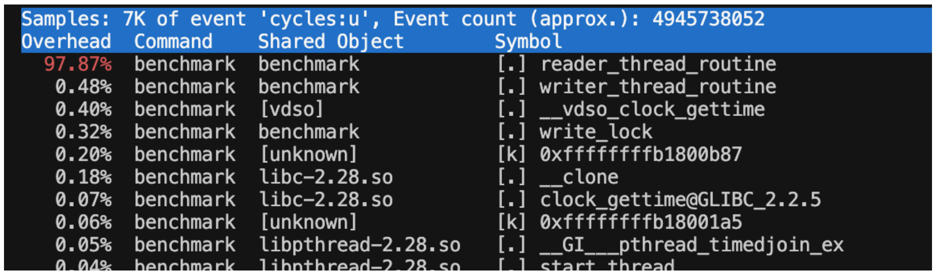
In this project, we effectively identified and mitigated a contention hotspot within the reader-writer lock, demonstrating an understanding of concurrency challenges and their solutions.

Through initial profiling using the Linux tool perf, we precisely located the performance bottleneck in the form of excessive contention primarily before the while (rwlock->writer); line within the read_lock function (45% overhead), particularly prominent in read-heavy workloads. We observed that sharing a single atomic variable caused a lot of contention among readers.

By implementing a distributed counter approach, allocating each read thread its own counter within an array, we significantly reduced contention among reader threads. We can see that the writer routine overhead decreased significantly and was marked as green in the report (3.25%) due to the fact that the writer had to wait less for the readers to decrease their distributed counters. This approach markedly enhanced parallelism, allowing readers to operate independently reducing contention or interference, while still preserving writer priority (Reduced original hotspot to 18%). We further optimized cache performance by choosing an array size of 16 on an 8-core machine, effectively reducing cache contention and false sharing. Subsequent benchmark tests (reduced mean-time and max-time) validated these improvements, showcasing reduced contention, improved parallelism, and sustained writer priority, affirming the enhanced efficiency and scalability of the database management system.

Some more Observations:

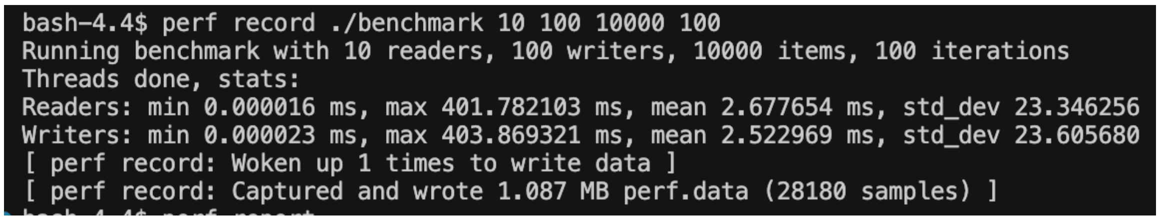
1) As the ratio between number of readers to writers in the benchmark increase, we observe that implementing a distributed counter almost eliminates any overhead in the writer routine as the reader contention decreases a lot.



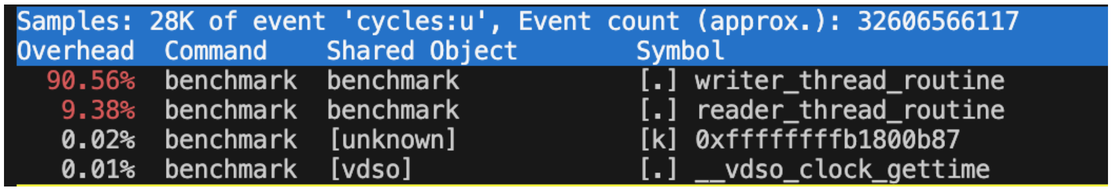
```
Samples: 7K of event 'cycles:u', Event count (approx.): 4945738052
Overhead  Command    Shared Object         Symbol
  97.87%  benchmark  benchmark             [.] reader_thread_routine
   0.48%  benchmark  benchmark             [.] writer_thread_routine
   0.40%  benchmark  [vdso]                [.] __vdso_clock_gettime
   0.32%  benchmark  benchmark             [.] write_lock
   0.20%  benchmark  [unknown]             [k] 0xffffffffb1800b87
   0.18%  benchmark  libc-2.28.so          [.] __clone
   0.07%  benchmark  libc-2.28.so          [.] clock_gettime@GLIBC_2.2.5
   0.06%  benchmark  [unknown]             [k] 0xffffffffb18001a5
   0.05%  benchmark  libpthread-2.28.so    [.] __GI___pthread_timedjoin_ex
   0.04%  benchmark  libpthread-2.28.so    [.] start_thread
```

Read overhead increases a lot

2) Our implementation doesn't scales well for a write heavy workload due to single write resource contention (multiple writes competing):



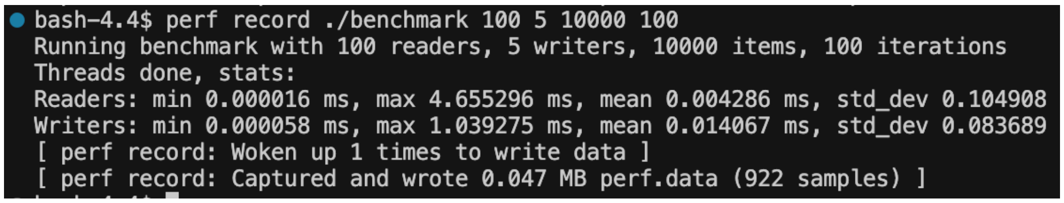
```
bash-4.4$ perf record ./benchmark 10 100 10000 100
Running benchmark with 10 readers, 100 writers, 10000 items, 100 iterations
Threads done, stats:
Readers: min 0.000016 ms, max 401.782103 ms, mean 2.677654 ms, std_dev 23.346256
Writers: min 0.000023 ms, max 403.869321 ms, mean 2.522969 ms, std_dev 23.605680
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.087 MB perf.data (28180 samples) ]
```



```
Samples: 28K of event 'cycles:u', Event count (approx.): 32606566117
Overhead  Command    Shared Object         Symbol
  90.56%  benchmark  benchmark             [.] writer_thread_routine
   9.38%  benchmark  benchmark             [.] reader_thread_routine
   0.02%  benchmark  [unknown]             [k] 0xffffffffb1800b87
   0.01%  benchmark  [vdso]                [.] __vdso_clock_gettime
```

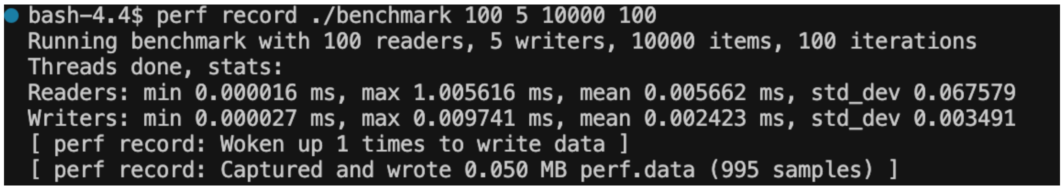
Writer thread routine shoots up (Contention on waiting on other writer threads)

3) Reducing counter array to 8 increases overheads for the reader as contention increases for the threads, making the array 32 doesn't affect the performance much in our case as only 16 positions are occupied.



```
bash-4.4$ perf record ./benchmark 100 5 10000 100
Running benchmark with 100 readers, 5 writers, 10000 items, 100 iterations
Threads done, stats:
Readers: min 0.000016 ms, max 4.655296 ms, mean 0.004286 ms, std_dev 0.104908
Writers: min 0.000058 ms, max 1.039275 ms, mean 0.014067 ms, std_dev 0.083689
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.047 MB perf.data (922 samples) ]
```

Num_counters = 8



```
bash-4.4$ perf record ./benchmark 100 5 10000 100
Running benchmark with 100 readers, 5 writers, 10000 items, 100 iterations
Threads done, stats:
Readers: min 0.000016 ms, max 1.005616 ms, mean 0.005662 ms, std_dev 0.067579
Writers: min 0.000027 ms, max 0.009741 ms, mean 0.002423 ms, std_dev 0.003491
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.050 MB perf.data (995 samples) ]
```

Num_counters = 32