[Multicore and GPU Programming] HW #2 Report

Introduction & Backgrounds

In HW #2, we need to optimize read/write performance of a thread-safe hash table which uses linear probing to deal with hash collisions, by mainly modifying the given naive `read` and `insert` implementation. Let's take a look at the overall structure of our hash table.

```C++
class hash_table {
  public:
    virtual uint32_t hash(uint32_t key) = 0;
    virtual uint32_t hash_next(uint32_t key, uint32_t prev_index) = 0;
    virtual bool read(uint32_t key, uint64_t* value_buffer) = 0;
    virtual bool insert(uint32_t key, uint64_t value) = 0;
    virtual int num_items() = 0;
    // virtual bool remove(uint32_t key) = 0; //JL: We do not consider remove function.
};
```

Listing 1: A base class for the hash table

```C++
class better_locked_probing_hash_table : public hash_table {
  private:
    Bucket* table;
    const int TABLE_SIZE; // we do not consider resizing.
    std::mutex global_mutex;
    // ...
  public:
    better_locked_probing_hash_table(int table_size)
      : TABLE_SIZE(table_size) {
      this->table = new Bucket[TABLE_SIZE]();
      for(int i=0; i < TABLE_SIZE; i++) {
        this->table[i].valid = 0; // means empty
      }
    }
    // ...
}
```

Listing 2: The given template for hash table implementation

```cpp
struct Bucket {
  uint32_t key;
  uint16_t valid; // 0=empty 1=occupied
  uint16_t meta;
  uint64_t value;
};
```

Listing 3: An entry for the hash table

Essentially, the hash table has a fixed-size array of `Bucket` s. Our job is to map a given key to an index of this array with hashing and linear probing in case of hash collision. There are some notable properties of this hash table to focus on: we need not consider resizing and remove operation. So we can ignore concurrency issues related to resizing or remove operation[1].

Based on these interfaces, the naive version simply uses one global mutex to lock the whole table for each `read` and `insert` operation, which means the entire scope of each operation is a critical section. In consequence, only one thread can `read` or `insert` at a time. This approach is trivially thread-safe, but is it the best approach?

Optimization Teqniques & Skills

The following teqniques are what I've tried and applied to optimize our hash table.

1) Fine-grained locks
2) Wait-free read
3) Spinlocks for critical sections
4) Prefetching buckets
5) Inlining virtual functions

1) Fine-grained locks

We can set fine-grained lock per a `Bucket` rather than depending on one global coarse-grained lock. Since simultaneous memory writes on $table[i]$ and $table[j]$ do not suffer from race condition for distinct integers i and j such that $i, j \in [0, TABLE\_SIZE)$, we can use different locks for different `Bucket` s to block access to a specific `Bucket` in question.

2) Wait-free read

Since our hash table is not restricted by any policies on `read` and `write`[2], we can simply remove locking & unlocking codes in `read` . By removing them, our hash table does not

[1] If we had to consider resizing, we would have to lock the entire table while copying elements to newly allocated memory. Similarly, if we had to consider the remove operation, we would have to carefully synchronize it with other operations.

[2] If there were some policies (i.e. "The memory write by the thread that started the insert later must overwrite the memory write by the thread that started the insert first.") which affect mechanism, we might need to introduce lock & unlock again.

guarantee that the value retrieved by a `read` will be the value at a specific point in time, but it eliminates the overhead associated with locking and unlocking during `read` s.

However, we need to handle a edge case caused by non-atomcity of mutating a `Bucket` . As we can see at line 17 in <Listing 8>, there is a software memory barrier `asm volatile ("" ::: "memory")` to prevent compiler's instruction reordering. Without this barrier, it is possible for g++ to reorder 3 mutations on a `Bucket` , so `b.valie = true` could be executed first. Why is this a problem?

Imagine a situation that Thread 0's `read` and Thread 1's `insert` run in parallel with `b.valid = true` assignment reordered to the top in the `if (!b.valid)` block.

```cpp
// <Listing 7> line 8-13
while (table[index].valid) {
  Bucket &b = table[index];
  if (b.key == key) {
    *value_buffer = b.value;
    return true;
  }
// ...
```

```cpp
// <Listing 8> line 14-18
// modified: move b.valid to top
if (!b.valid) {
  b.valid = true;
  b.key = key;
  b.value = value;
}
// ...
```

In this case, if Thread 0 executes the code on the left after Thread 1 has completed execution up to `b.valid = true` , Thread 0 would enter the while loop even though the key and value have not been yet set yet. Once entered inside the loop, it must continue the loop until `probe_count` reaches `TABLE_SIZE` .

We have solved the theoretical problem caused by compiler reordering. But what about memory reordering by a hardware? Modern AMD64[3] CPUs implement out-of-order execution ([1] Advanced Micro Devices Inc. 2024, 187-188) to improve performance. However, this out-of-order execution behavior is restricted by policies made by manufacturers to guarantee the correctness of a program. In our case, since stores do not pass previous stores (out-of-order writes are not allowed) ([1] Advanced Micro Devices Inc. 2024, 188-189), we need not add explicit hardware write barriers between these mutations.

3) Spinlocks for critical sections

There are two fundamental building blocks to ensure that only one thread runs critical sections: spinlocks and mutexes. The main difference between them is that mutexes cause the thread in question to sleep, whereas spinlocks keep the thread actively spinning in a loop until a certain condition is met, without sleeping.

Since our critical sections are relatively short, consisting of at most three assignments, we can benefit from using spinlocks which eliminate the overhead associated with sleeping and waking. Additionally, with a sufficient number of detected cores (128 cores in our case, checked with `lscpu`) to accommodate 8 threads, we generally do not suffer from starvation.

[3]Our server uses AMD EPYC 7452 32-core processor (checked with `lscpu`).

```
$ lscpu
  Architecture:                    x86_64
  CPU op-mode(s):                  32-bit, 64-bit
  Byte Order:                      Little Endian
  Address sizes:                   43 bits physical, 48 bits virtual
  CPU(s):                          128
  On-line CPU(s) list:             0-127
  Thread(s) per core:              2
  Core(s) per socket:              32
  Socket(s):                       2
  NUMA node(s):                    2
  Vendor ID:                       AuthenticAMD
  CPU family:                      23
  Model:                           49
  Model name:                      AMD EPYC 7452 32-Core Processor
```

Listing 4: The output of lscpu

To implement spinlocks, we can use C++'s `volatile` keyword in conjunction with GCC's `__atomic_test_and_set` with memory ordering `__ATOMIC_ACQUIRE`, and `__atomic_clear` with memory ordering `__ATOMIC_RELEASE`.

`volatile` simply prevents GCC from optimizing out a specified variable used for implementing a conditional loop in spinlocks.

`__atomic_test_and_set` in conjunction with `__ATOMIC_ACQUIRE` atomically tests the previous contents and sets nonzero "set" value. The return value is `true` iff the previous contents were "set". The specified memory ordering guarantees instructions above here are not be reordered crossing this boundary ([3] Free Software Foundation. 2024).

`__atomic_clear` in conjunction with `__ATOMIC_ACQUIRE` clears the bits while guaranteeing instructions below here are not be reordered crossing this boundary ([3] Free Software Foundation. 2024).

4) Prefetching buckets

Since memory accesses to each `Bucket` are fairly unpredictable due to key mapping by the hash function's effective distribution of keys, we can benefit from explicitly instructing the hardware to prefetch data. This ensures that the actual data access hits caches rather than DRAM.

```
1      __builtin_prefetch(&table[index], 0);
2  53d6: 48 8b 45 d8           mov    rax,QWORD PTR [rbp-0x28]
3  53da: 48 8b 40 08           mov    rax,QWORD PTR [rax+0x8]
4  53de: 48 8b 55 f0           mov    rdx,QWORD PTR [rbp-0x10]
5  53e2: 48 c1 e2 04           shl    rdx,0x4
6  53e6: 48 01 d0              add    rax,rdx
7  53e9: 0f 18 08              prefetcht0 BYTE PTR [rax]
```

Listing 5: A generated prefetch instruction

AMD EPYC 7452 32-core processor, our server's CPU, supports prefetch instructions. So before accessing some data, we can move it into all cache levels ([2] Advanced Micro Devices Inc. 2024, 285).

5) Inlining virtual functions

The last thing we can try is substituting `this->hash` and `this->hash_next` functions calls with its body. In C++, which virtual methods are called is determined dynamically, and calling them involves referencing the generated virtual tables. Since `this->hash` and `this->hash_next` can be easily inlined, we can effectively avoid these overheads.

Evaluation

The followings are the final version of my modifications on `better_locked_probing_hash_table`.

```C++
class better_locked_probing_hash_table : public hash_table {
  private:
    Bucket *table;
    const int TABLE_SIZE;
    std::mutex global_mutex;

    /* TODO: put your own code here  (if you need something)*/
    /*****************/
    volatile bool locks[5000000] = {0};
    /*****************/
    /* TODO: put your own code here */
```

Listing 6: A new member variable for locking each Bucket

```cpp
bool read(uint32_t key, uint64_t* value_buffer) {
  key = ((key >> 16) ^ key) * 0x45d9f3b;
  key = ((key >> 16) ^ key) * 0x45d9f3b;
  uint64_t index = ((key >> 16) ^ key) % TABLE_SIZE;
  __builtin_prefetch(&table[index], 0);
  int probe_count = 0;

  while (table[index].valid) {
    Bucket &b = table[index];
    if (b.key == key) {
      *value_buffer = b.value;
      return true;
    }
    probe_count++;
    index = (index + 1) % TABLE_SIZE;
    if (probe_count >= TABLE_SIZE)
      return false;
  }
  return false;
}
```

Listing 7: An optimized read method implementation

```cpp
bool insert(uint32_t key, uint64_t value) {
  key = ((key >> 16) ^ key) * 0x45d9f3b;
  key = ((key >> 16) ^ key) * 0x45d9f3b;
  uint64_t index = ((key >> 16) ^ key) % TABLE_SIZE;
  __builtin_prefetch(&table[index], 1);
  int probe_count = 0;

  for (;;) {
    while (__atomic_test_and_set(&locks[index], __ATOMIC_ACQUIRE)) {
      while (locks[index]) {}
    }

    Bucket &b = table[index];
    if (!b.valid) {
      b.key = key;
      b.value = value;
      asm volatile ("" ::: "memory");
      b.valid = true;
      __atomic_clear(&locks[index], __ATOMIC_RELEASE);
      return true;
    } else if (b.key == key) {
      b.value = value;
      __atomic_clear(&locks[index], __ATOMIC_RELEASE);
      return true;
    }
    __atomic_clear(&locks[index], __ATOMIC_RELEASE);
    probe_count++;
    index = (index + 1) % TABLE_SIZE;
    if (probe_count >= TABLE_SIZE)
      return false;
  }
}
```

Listing 8: An optimized insert method implementation

The following table shows the benchmark results with different approachs for thread-safe hash table implementation (where table_size = 5000000, num_init_items = 2000000, num_new_items = 2000000, additional_reads_per_op = 9, num_threads = 8). Each benchmark result is a simple mean of 10 samples.

	init	test
Given naive implementation	2.10979 sec	5.86671 sec

	init	test
Wait-free read & fine-grained lock with mutexes	0.09428 sec	0.25584 sec
Wait-free read & fine-grained lock with spinlocks	0.06321 sec	0.21565 sec
Wait-free read & fine-grained lock with spinlocks & prefetching	0.06135 sec	0.23430 sec
Wait-free read & fine-grained lock with spinlocks & inlined virtual functions	0.05774 sec	0.20018 sec
Wait-free read & fine-grained lock with spinlocks & prefetching & inlined virtual functions	0.04792 sec	0.18957 sec

The interesting point here is that prefetching only showed noticeable effects when used with virtual function inlining. This might be related to the caching of vtables as well the cache eviction policies of the hardware, but the exact reason should be derived by using profiling tools like `perf`.

Conclusion

By using 1) Fine-grained locks, 2) Wait-free read, 3) Spinlocks for critical sections, 4) memory prefetching, and 5) virtual function inlining, we could further improve our hash table's read/write performance from (init: 2.10979, test: 5.86671) to (init: 0.04792, test: 0.18957).

However, there is no silver bullet for optimization teqniques. We always need to benchmark, and should take care of hardware-specific behaviors, compilers, and most importantly our algorithms to fully optimize our implementation.

References

[1] Advanced Micro Devices Inc. "AMD64 Architecture Programmer's Manual Volume 2: System Programming" (2024). Available at https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. Accessed 2024-04-27.

[2] Advanced Micro Devices Inc. "AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions" (2024). Available at https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf. Accessed 2024-04-27.

[3] Free Software Foundation. "A GNU Manual" (2024). Available at https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html. Accessed 2024-04-28.