École polytechnique fédérale de Lausanne

Optional Semester Project

Verified double-hashing hash map

Martin Vassor

supervised by
K. Argyraki, Pr.
G. Candea, Pr.
A. Zaostrovnykh, Ph. D. Student

January 13, 2017

1 Introduction

This report explain, in an informal way, the implementation and verification of a double-hash hash map. It does not aim to provide a complete and detailed explication of each line of code. Instead, the goal is to synthesize the main points needed to understand both the code and the verification.

The actual verification contains more than 100 new lemmas and fixpoints, and it would make no sense to formally describe each of them here, as a formal definition and proof are provided. Also, most of those are trivial proofs and the name is explicit enough to understand their behaviour.

Thus, this document is more intended to be a companion to understand the actual proof.

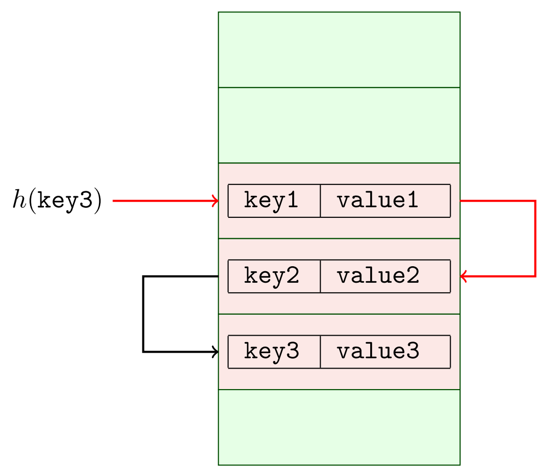
2 Implementation

2.1 Provided implementation

The implementation I was provided was a naive hash map, in which a $< key, value >$ tuple is inserted at the first free cell after $h(key)$, where h is a given hash function.

Thus, in case of multiple conflicts, the same cells will be tested. For instance, in Example 1, if $h(key1) = h(key2) = h(key3)$, then there is 2 unsuccessful accesses before finding an empty cell to insert $key3$ in.

Example 1: Multiple conflicts when inserting in a naive hash map.

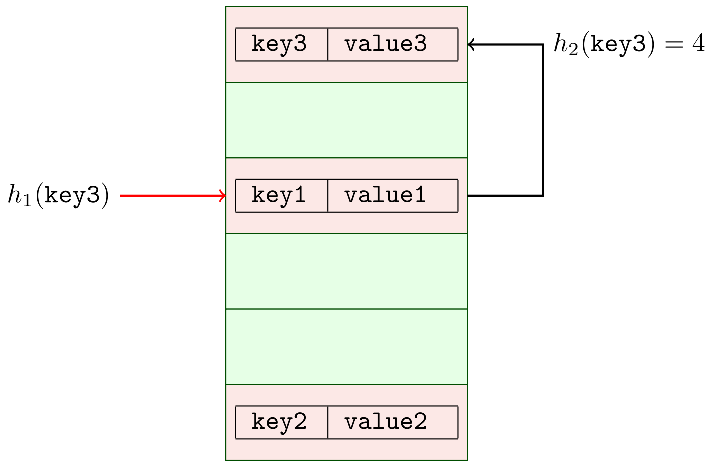


2.2 Double-hash implementation

The solution implemented during this project is *double-hashing*. In double-hashing, instead of searching in the following cell in case of conflict, the key is re-hashed using a second hash-function. This second hash-function determines an offset, and after each unsuccessful try, the $current\_index + offset$-th cell is looked-up.

For instance, Example 2 shows the same accesses as the previous example, but with double-hashing (the first hash-function being the same). As $key2$ and $key3$ have different second hashes, their second choice cell is not the same. Then inserting $key3$ only conflicts with $key1$.

Example 2: Multiple conflicts when inserting in a double-hash hash map.



2.3 Benchmark

Two cases have been tested: the general case, where any value can been searched for. The worst case been trying to access a value not present (this requires to go through the whole array), a second realistic case have been tested, where the values searched for where in the map.

The value were compared to the naive implementation and with two C++ implementations using the standard `unordered_map` data structure. The first C++ version is one using the default hash function, the second using the same hash function as in the C versions.

The test consist of uniformly distributed accesses (among all possible keys in the general case, and among existing keys in the second one). The probability of insertion/deletion is determined in function of the current load and the target load.

The following subsections show the results for both cases, for 70% of read accesses. It appears that the percentage of read accesses does not influence “the shape” of the result. Results for other read proportion as well as raw data are available on the github repository.

The code was compiled with GCC 4.9.3. It appears that bug as been reported for version greater than 4.6.2: performances of `unordered_map` are 3× slower compared to 4.6.2[1].

[1] `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=54075`

2.3.1 General case

Results are presented in Figure 1. For this test case, 10000 accesses were performed. The interesting point is that the double hash implementation perform better with higher loads. This behaviour is due to the fact that the probability that a key is present increases with load. Hence, in an execution at higher load, there are less misses than at low loads.

At low loads, the naive implementation performs better than the double hash one. This is a good illustration of the locality problem: in the naive implementation, accesses are performed in order. Then, both spatial locality of cache lines and address prediction works better. An evaluation with cachegrind shows the difference (see Example 3).

Example 3: Cache accesses

Cachegrind results for load = 10%, 70% of read accesses, first for the naive implementation:

```
D   refs:        319,195,365  (297,779,797 rd   + 21,415,568 wr)
D1  misses:       27,235,744  ( 27,203,118 rd   +     32,626 wr)
LLd misses:           10,861  (      5,701 rd   +      5,160 wr)
D1  miss rate:          8.5% (         9.1%     +        0.1%  )
LLd miss rate:          0.0% (         0.0%     +        0.0%  )
```

And for the double hash implementation:

```
D   refs:        328,259,239  (328,032,233 rd   + 227,006 wr)
D1  misses:      303,932,169  (303,902,705 rd   +  29,464 wr)
LLd misses:           11,745  (      4,032 rd   +   7,713 wr)
D1  miss rate:         92.5% (        92.6%     +    12.9%  )
LLd miss rate:          0.0% (         0.0%     +     3.3%  )
```

In this example, there are more than 90% of data cache misses in L1 for the double hash implementation, while the naive one achieve less than 10%. In the double hash implementation, almost all accesses hit on the LLC.

2.3.2 Access only existing keys

In the case all accessed keys exist, double hash implementation is approximatively an order of magnitude faster than C++, on GCC 4.9.3. However, some performance issues have been reported for `unordered_map` higher than 4.7.1 (3× slower than 4.6.2). Hence, this performance evaluation should be performed again with a lower version for a fair comparison.

However, compared to the naive C implementation, the results are quite good: the double hash implementation is 2 orders of magnitude faster than the naive one.

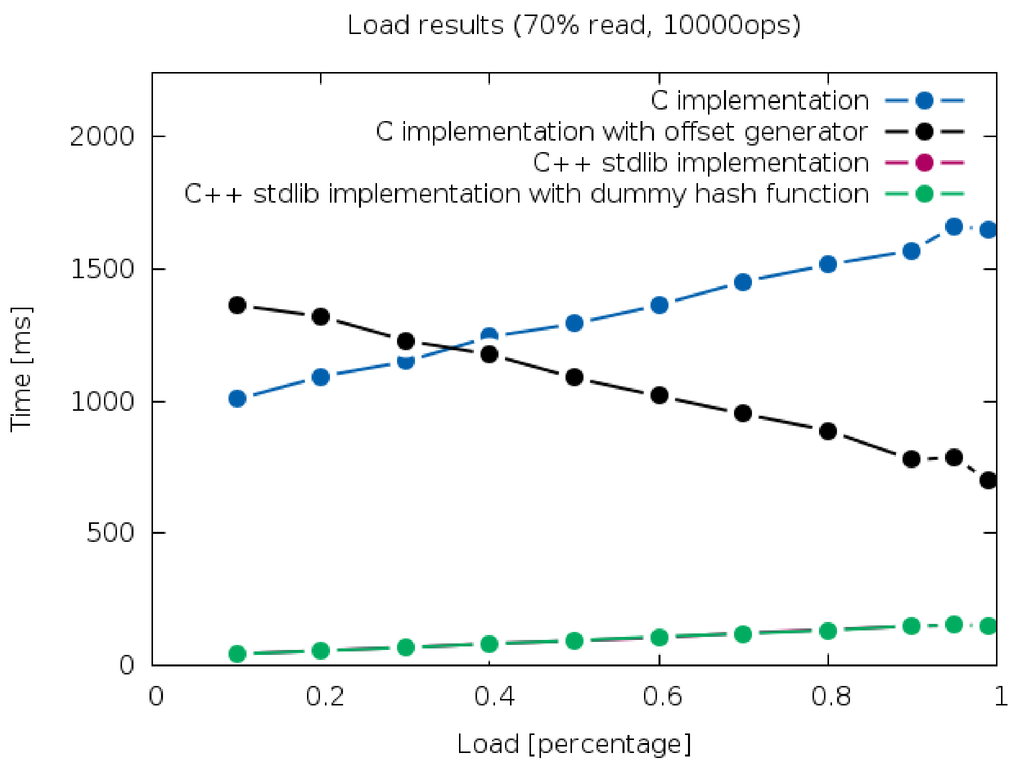


Figure 1: Timings for 70% read accesses, including non existing keys.

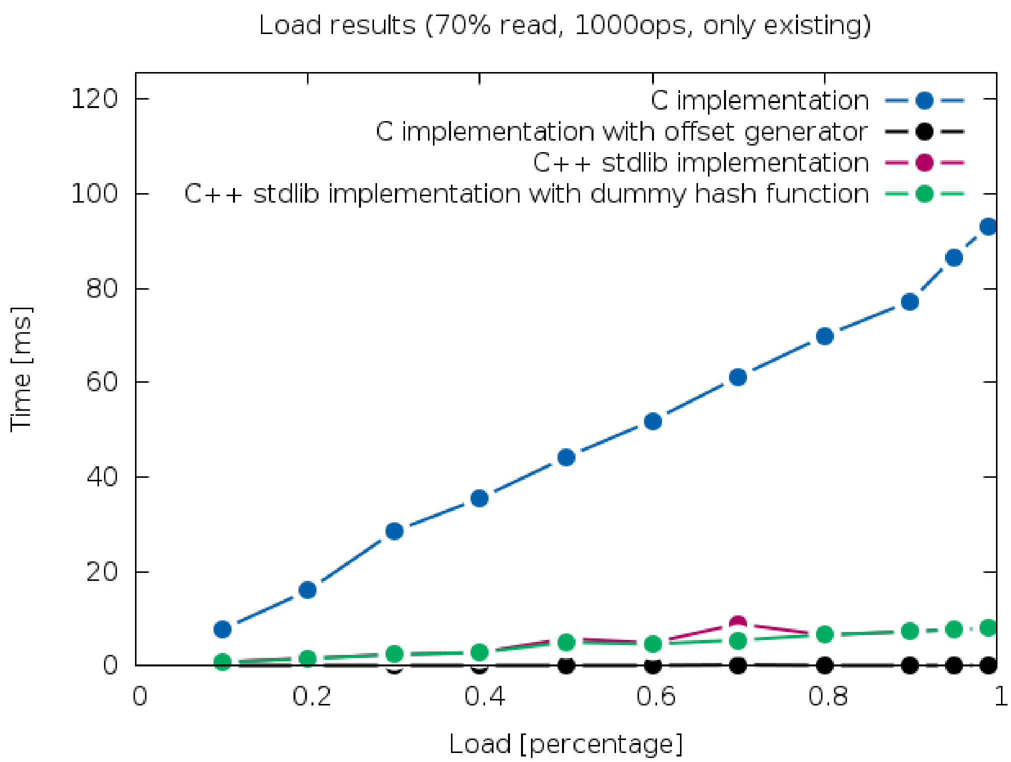


Figure 2: Timings for 70% read accesses, excluding non existing keys.

3 Verification

3.1 Provided proof

3.1.1 Requirement (R)

Figure 3 present the relevant part of the proof of the original loop in `find_key`. The last statement (`no_key_found(ks, k)`) requires the property `not_my_key(k)` to be verified for all current keys in the mapping, ensured by the `up_to(nat_of_int(length(ks)), ...(not_my_key)(k)...)` statement.

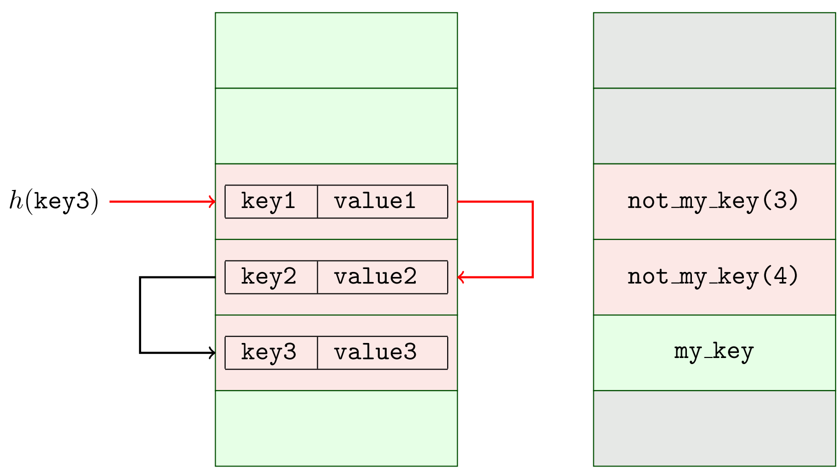
`up_to(nat, prop)` verifies that `prop` is ensured for all `i` below `nat`:
`up_to(0, prop) = true`
`up_to(n, prop) = prop(n-1) && up_to(n-1, prop)`

This `up_to` statement is proved by the *for*-loop invariant: at each round, `not_my_key(k, nth(index, ks))` is ensured, either because the cell is empty (`no_busy_no_key` lemma), either because the hash does not match (`no_hash_no_key` lemma), either because the key does not match (hence inferred by Verifast).

Finally, the *for*-loop only proves that the `up_to` statement holds when starting from `index = start` and looping. The lemma `by_loop_for_all` prove that this loop access is equivalent to a continuous access from `0` to `length`.

Example 4 represent a successful search in the map. The search starts at index $h(key3)$. As long as the key is not found at index i, `not_my_key(i)` is asserted. Finally, when $key3$ is found, it is ensured to be the right key and returned.

Example 4: Successful search

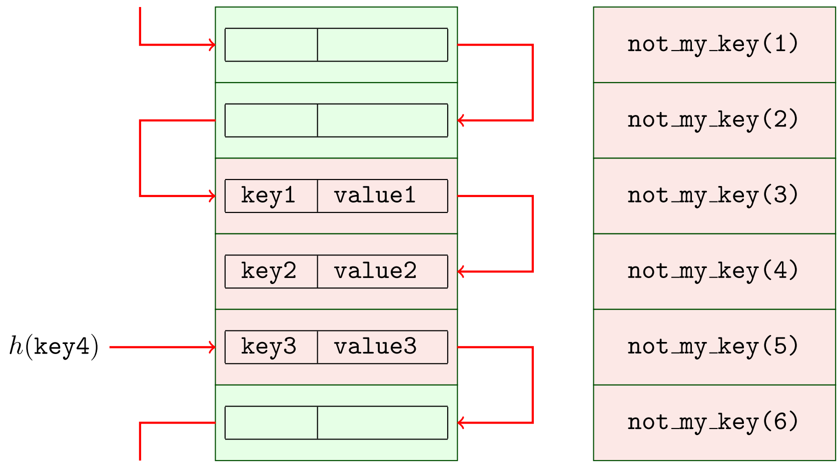


In case of unsuccessful search, as in Example 5, `not_my_key(i)` is asserted for all indexes, ensuring that the key is not present in the map.

Hence, an invariant of the *for*-loop is that `not_my_key` is asserted *for all indexes from start up to* i, by ring accesses, i being the loop iterator.

Example 5

`not_my_key` is ensured both when the cell is empty (green cell in this example), or when the cell is busy, but occupied by an other key (a busy cell is in red in this example).



3.1.2 Impact of the modifications

The modifications have two main impacts: first, the accesses are not performed in the same order. The second impact is that the specification is not true any more in the general case.

Access order: As explain above, with double hashing, cells of the map are not accessed by loop any more. Hence, the `by_loop_for_all` lemma doesn't apply any more. Let *stripe* be the function which, given a loop iteration, returns the index of the cell looked-up at this iteration (parametrized by *start*, *step* and *capacity*). This problem is solved by computing the antecedent of each cell.

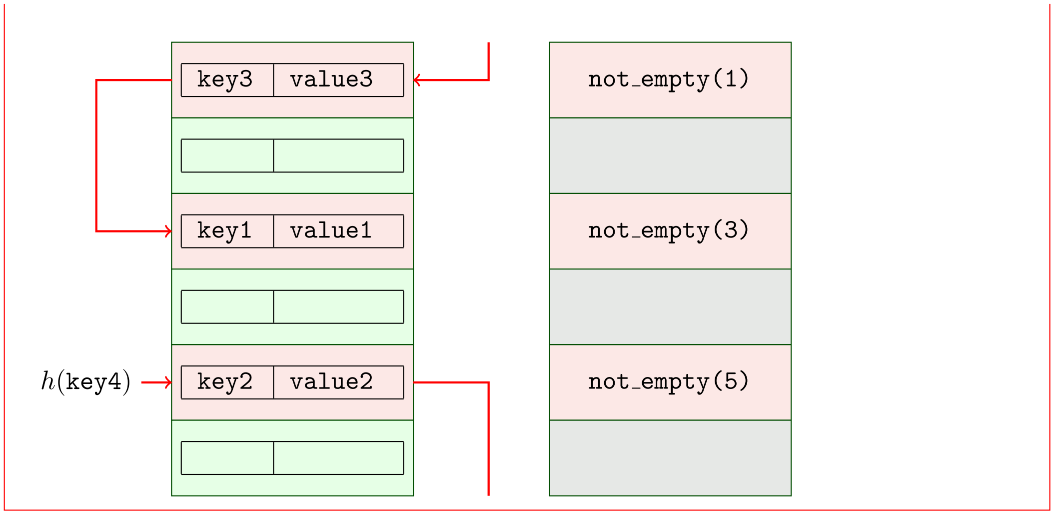
Hence, at iteration i, for any cell *map*[`index`], if the antecedent of *map*[`index`] is less than i, then `not_my_key(index)` is ensured.

The new way to ensure `not_my_key` for all indexes is then to ensures that all index has an antecedent w.r.t. the *stripe* function.

New requirements: However, it is not always the case that every cell is reached. An example is provided in Example 6. Actually, after the Chinese remainder theorem, the *step* and the *capacity* must be coprime in order to ensure that every cell is eventually tested.

Hence, this coprimeness is a new requirement of the specification. Technically, it is sufficient to have a capacity being a power of 2 and to have only odd $offset$ hashes.

Example 6



3.2 The `stripe_l_fp` fixpoint

3.2.1 Definition

First, a fixpoint is defined, which returns the index to be updated after n iterations with an offset of *step*, starting from *start* with capacity *capa*.

A lemma ensuring that `stripe` $= start + n * step\%capa$ is proved.

Definition 1: stripe(int start, int step, nat n, int capa)

```
fixpoint int stripe(int start, int step, nat n, int capa) {
  switch(n) {
    case zero: return start;
    case succ(m): return
      (stripe(start, step, m, capa) + step) % capa;
  }
}
```

The `stripe_l_fp` fixpoint builds a `list<option<nat>>` given a starting point, an offset, a number of accesses and a capacity. The base case of this fixpoint is to generate a `list` containing only `nones` (fixpoint `gen_none`), if `zero` accesses are performed. The recursive case is to update the $start + n * offset\%capa$ cell, using the above `stripe` fixpoint.

Definition 2: stripe_l_fp(int start, int step, nat bound, int capa)

The `update (index, elem, list)` fixpoint returns `list` with the `index`-th element updated to `elem`.

```
fixpoint list<option<nat> > stripe_l_fp(int start,
  int step, nat n, int capa)
{
```

```
    switch(n) {
        case zero: return gen_none(nat_of_int(capa));
        case succ(m): return
            update(stripe(start, step, n, capa), some(n),
                stripe_l_fp(start, step, m, capa));
    }
}
```

Example 7: `stripe_l_fp(0, 2, 5, 7)`

Calling `stripe_l_fp(0, 2, 5, 7)` produces the following list. Notice that the base case returns a list containing only `none`s, not a list containing a `some(0)`.

`none`
`some(4)`
`some(1)`
`some(5)`
`some(2)`
`none`
`some(3)`

3.2.2 Properties

The main property required is that the number of cell containing `some(i)` (for any `i`) is equal to the number of steps done. This property will later be used to ensures that all cells are eventually reached (see Subsection 3.2.4). The function that count the number of such cells is named `count_some(list<option<nat>> list)`.

There are also other properties which are used internally to prove the `count_some` property. The main lemma is named `stripe_l`.

Lemma 1: Prototype of `stripe_l`

`list_contains_stripe` ensures that if a cell contains `some(i)`, then `i` is the antecedent of the cell index w.r.t. stripe.

```
lemma list<option<nat> > stripe_l(int start, int step, nat n,
   int capa)
requires 0 <= start &*& start < capa &*& step > 0
   &*& n <= capa &*& coprime(step, capa) &*& step < capa;
ensures count_some(result) == n
   &*& length(result) == capa
   &*& true == up_to(nat_of_int(capa),
      (list_contains_stripes)(result, start, step))
   &*& true == up_to(nat_of_int(capa),
      (lst_opt_less_than_n)(result, n))
   &*& true == forall(result, opt_not_zero)
   &*& result == stripe_l_fp(start, step, n, capa)
   &*& coprime(step, capa);
```

3.2.3 Proof of `stripe_l`

The proof of these properties relies on the fact that the same cell is not updated twice. Once this is ensured, the construction of the fixpoint ensures the validity of the properties.

Algorithm 1 shows the main steps of the proof. In the base case, all trivially holds. In the inductive case, if the `stripe(start, step, n, capa)`-th cell (i.e. the one hit at n-th iteration) already contains `some(i)`, the `list_contains_stripes` property ensures that `stripe(start, step, i, capa)`-th cell is the one we are hitting. Hence, $start + step \times i \% capa = start + step \times n \% capa$, with $n - i < capa$. Then the Chinese remainder theorem leads to a contradiction.

3.2.4 From `stripe` fixpoint to `R`

3.3 Proof of the *Chinese remainder theorem*

3.3.1 Properties

The goal of the *Chinese remainder theorem* is to highlight a contradiction in the `stripe_l` proof. We have that $diff \times step \% capa = 0$. The contradiction we want to highlight is that in the given environment, $diff$ can only be 0, i.e. the supposed previous value is the same that the one we want to write, that is the n-th iteration is supposed to be already written to the list.

This is reduced to the following lemma: if $x \% n1 = 0$, $x \% n2 = 0$, $n1$ and $n2$ are coprime, and $x < n1 \times n2$, then $x = 0$. It is also required that $n1 > 0$, $n2 > 0$ and $x \geq 0$.

In `stripe_l`, this is applied to $x = diff \times$ `step`, $n1 =$ `step` and $n2 =$ `capa`.

This lemma is a direct consequence of the *uniqueness* property of the *Chinese remainder theorem*. Although it is simple to show informally, Verifast first requires to build *gcd* which is quite long. All the proof is done in a separate file `Chinese_remainder_th.gh`. The proof takes around 1000 lines of code (which less than 200 are *assert*-s or comments and could be remove).

Input: int start, int step, nat n, int capa

```
switch n do
    case zero
        // All hold by construction
    end
    case succ(m)
        // Recursive call, the termination is ensured by n > m
        list lst ⟵ stripe_l(start, step, m, capa)
        // Now, we want to update the stripe(start, step, n, capa)-th to
           some(n)
        // Proof by contradiction that the cell contains none
        switch nth(stripe(start, step, n, capa), lst do
            case some(i)
                assert start + i × step%capa = start + n × step%capa;
                assert (n − i) × step%capa = 0;
                assert n − i < capa;
                // The chinese remainder theorem applies and shows a
                   contradiction
                chinese_remainder_theorem(step, capa, (n − i) × step);
            end
            case none
            end
        endsw
        // We now that the stripe(start, step, n, capa)-th cell contains
           a none, which we update to some(n), so the properties hold
           for the updated list.
        return update(stripe(start, step, n, capa), some(n), lst)
    end
endsw
```

$n - i$ is noted $diff$ in the following parts

Algorithm 1: Proof of `stripe_l`

3.3.2 Proof by contradiction

In Verifast, the proof of the `bin_chinese_remainder_theorem` lemma is quite long (approx. 300 lines). However, most of it is only arithmetic statements. Hence, informally, the proof is much shorter. Algorithm 2 sketches the main cases. The main part (*if* $x > 1$ branch) decompose x into $n1 \times k1 = n2 \times k2$. After justifying why $k1\%n2 \neq 0$, it considers $gcd(k1, n2) = a$, which can not be 1. Then remaining case ($a \neq 1$, $k1\%n2 \neq 0$, $x > 1$) calls recursively the theorem, on $\frac{n2}{a} = b$.

3.3.3 Assumed lemma

One lemma remains assumed:

Lemma 2: `gcd_mul`

```
lemma void gcd_mul(int n1, int n2, int n3)
requires coprime(n1, n3) &*& coprime(n2, n3);
ensures coprime(n1*n2, n3) &*& coprime(n1, n3)
   &*& coprime(n2, n3);
```

4 Conclusion

4.1 Work done

This semester project was two folds:

First, analysing the provided implementation to see which part could be improved. After that, I implemented an optimization, which, I later learn, is called *double hashing* (before learning that, I called it map with offset generator, which explain some naming, e.g. `map_generator.c`).

Second, formally proving that the map is correct. The semantic of *correctness* is the same as the previous one. The only difference in the specification is the requirement of coprime capacity and offset, which is easily achieved, for instance taking a capacity 2^n and an odd offset.

The formal proof has been completed, except for one lemma:

$$a \perp c \wedge b \perp c \Rightarrow (a \times b) \perp c$$

4.2 Validity of benchmark

The benchmark has been done to give an idea of the performance gain. Of course, performance evaluation could be much improved, with more time available. In particular, one could improve the following points:

- Test on different compilers, as GCC greater than 4.7.1 suffer from performance issues on `unordered map`.

if $x = 1$ **then**
  $x\%n1 = 0 \Rightarrow n1 = 1$;
  $x\%n2 = 0 \Rightarrow n2 = 1$;
  assert $n1 \times n2 = 1$;
  assert $x = n1 \times n2$;
  contradiction;
else if $x > 1$ **then**
  $x\%n1 = 0 \Rightarrow \exists k1 | n1 \times k1 = x$;
  $x\%n2 = 0 \Rightarrow \exists k2 | n2 \times k2 = x$;
  assert $k1 \neq 0$;
  if $k1\%n2 = 0$ **then**
    $\beta \longleftarrow k1/n2$;
    assert $\beta \times n2 = k1$;
    assert $\beta \geq 1$;
    assert $\beta \times n2 \leq k1$;
    assert $x = n1 \times k1 \geq n1 \times n2$;
    contradiction;
  else
    $a \longleftarrow gcd(k1, n2)$;
    $b \longleftarrow n2/a$, **assert** $b \neq 0$;
    $\gamma \longleftarrow k2/a$;
    assert $gcd(b, \gamma) = 1$;
    if $gcd(n1, b) \neq 1$ **then**
      assert $gcd(n1, a \times b) \neq 1$;
      assert $gcd(n1, n2) \neq 1$;
    end
    if $a = 1$ **then**
      assert $\gamma = k1 \wedge b = n2$;
      $gcd(b, \gamma) = 1 \wedge gcd(n1, b) = 1 \Rightarrow gcd(n1 \times \gamma, b) = 1$;
      contradiction $gcd(x, n2) = 1$;
    else
      `// The termination is ensured by b < n2`
      `bin_chinese_remainder_theorem`($n1$, b, $k2 \times b$);
      assert $k2 \times b = 0$;
      assert $k2 = 0$;
      contradiction $n2 \times k2 = x = 0$;
    end
  end
else
  assert $x = 0$;
end

Algorithm 2: Proof of `bin_chinese_remainder_theorem`

- Use an other distribution. Here, accesses are uniformly distributed. Other distribution such as Zipf's distribution might be more suitable. Another idea is to adapt some real program to this hash table to have real accesses.

4.3 Forthcoming work

Although most of the work is completed, some parts remain to do:

- The `gcd_mul` lemma is to be proved. Proving it might requires to prove prime factor decomposition in Verifast.
- Formally prove that a capacity of 2^n and an odd *offset hash* are coprime.

```
int i = 0;
for (; i < capacity; ++i)
/*@ invariant ... &*&
    true == up_to(nat_of_int(i),
        (byLoopNthProp)(ks, (not_my_key)(k),
            capacity, start));
@*/
//@ decreases capacity - i;
{
    int index = loop(start + i, capacity);
    int bb = busybits[index];
    int kh = k_hashes[index];
    void* kp = keyps[index];
    if (bb != 0 && kh == key_hash) {
        if (eq(kp, keyp)) {
            //@ hmap_find_this_key(hm, index, k);
            return index;
        }
    } else {
        //@ if (bb != 0) no_hash_no_key(ks, khs, k, index, hsh);
        //@ if (bb == 0) no_bb_no_key(ks, bbs, index);
    }
    //@ assert(true == not_my_key(k, nth(index, ks)));
}
/*@ by_loop_for_all(ks, (not_my_key)(k),
    start, capacity, nat_of_int(capacity));
@*/
/*@ assert true == up_to(nat_of_int(length(ks)),
    (nthProp)(ks, (not_my_key)(k)));
@*/
//@ no_key_found(ks, k);
```

Figure 3: Original *for*-loop for searching a key