JDK-8165674 states that the G1CMMarkStack::_out_of_memory member may be redundant, just mirroring the value of G1ConcurrentMark::_has_overflown.

Here is a prove that those fields are actually duplicates.

1. Look at the method "void G1CMMarkStack::reset_marking_state(bool clear_overflow = true);

This method never called with clear_overflow = false (and we can remove parameter clear_overflow from this method).

Now it is clear that any call to the reset_marking_state() will set both G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown to the false.

2. Both G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown are initialized to false.

3. The only code that that changes G1CMMarkStack::_out_of_memory to the true is at G1CMMarkStack::par_push_chunk(oop* ptr_arr) and it's only caller mark_stack_push(oop* arr) uses return value to set G1ConcurrentMark::_has_overflown to the true too.

4. Now we have only one place where values of G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown may become different:

    _cm->clear_has_overflown() inside of G1CMConcurrentMarkingTask::work(...).

5. There is one place where code assumes that values of G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown might be different - inside of G1ConcurrentMark::weakRefsWork(...) there is code

```
    if (_global_mark_stack.is_out_of_memory()) {
      // This should have been done already when we tried to push an
      // entry on to the global mark stack. But let's do it again.
      set_has_overflown();
    }
```

6. Let see these two "suspicious" (4. and 5.) pieces in the execution context:

```
void ConcurrentMarkThread::run_service() {
...
  for (uint iter = 1; true; ++iter) {
    if (!cm()->has_aborted()) {
      G1ConcPhaseTimer t(_cm, "Concurrent Mark From Roots");
      _cm->mark_from_roots(); /* here we call G1CMConcurrentMarkingTask::work(...)
                                 with code in question:
                                 do {
                                      ...
                                      the_task->do_marking_step(...);
                                      _cm->clear_has_overflown();
                                      ...
                                   } while (the_task->has_aborted());
                                */

    }
    ...

    if (!cm()->has_aborted()) {
      ...
      CMCheckpointRootsFinalClosure final_cl(_cm);
      VM_CGC_Operation op(&final_cl, "Pause Remark");
      VMThread::execute(&op);  /* here wee call ConcurrentMark::weakRefsWork(...)
                                  with code in question:
                                  ...
                                  if (_global_mark_stack.is_out_of_memory()) {
```

```
                                    set_has_overflown();
                                  }
                                  ...
                                */
      ...
    }

    if (!cm()->restart_for_overflow() || cm()->has_aborted()) {
      break;
    }
  } // end for loop
  ...
}
```

_cm->mark_from_roots() waits till marking complete or failed and VMThread::execute(&op) waits for completion (or failure) of VM_CGC_Operation.

So those two blocks are executed sequentially without interfering.

It means that value of G1CMMarkStack::_out_of_memory is not used in the context of mark_from_roots() - only at the very end of G1CMTask::do_marking_step() we have guarantee(!_cm->mark_stack_overflow(), "only way to reach here");
(mark_stack_overflow is delegated to the G1CMMarkStack::_out_of_memory) in case of successful completion of marking.

It means we can set G1CMMarkStack::_out_of_memory to the false at same location where we call _cm->clear_has_overflown() and will not change execution , because G1CMMarkStack::_out_of_memory not used under mark_from_roots() and will be false after success completion of mark_from_roots().

Now let look at

```
  if (_global_mark_stack.is_out_of_memory()) {
    set_has_overflown();
  }
```

We entered VM_CGC_Operation with both G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown equal to false and if under context of CMCheckpointRootsFinalClosurethreir values cannot be changed individually. So set_has_overflown() call is no-op here.

At the end of G1ConcurrentMark::checkpointRootsFinal() we call reset_marking_state() in case of failure (and go back to the mark_from_roots()) and set_non_marking_state() in case of success. It means that mark_from_roots() always starts with mark_from_roots() with G1CMMarkStack::_out_of_memory and G1ConcurrentMark::_has_overflown equal to false.

So we proved redundancy stated in the JDK-8165674