Findings

```
void GenCollectedHeap::process_roots(ScanningOption so,
                                     OopClosure* strong_roots,
                                     CLDClosure* strong_cld_closure,
                                     CLDClosure* weak_cld_closure,
                                     CodeBlobToOopClosure* code_roots) {
  // General roots.
  assert(code_roots != NULL, "code root closure should always be set");

  ClassLoaderDataGraph::roots_cld_do(strong_cld_closure, weak_cld_closure);

  // Only process code roots from thread stacks if we aren't visiting the entire CodeCache
anyway
  CodeBlobToOopClosure* roots_from_code_p = (so & SO_AllCodeCache) ? NULL :
code_roots;

  Threads::oops_do(strong_roots, roots_from_code_p);

  OopStorageSet::strong_oops_do(strong_roots);

  if (so & SO_ScavengeCodeCache) {
    assert(code_roots != NULL, "must supply closure for code cache");

    // We only visit parts of the CodeCache when scavenging.
    ScavengableNMethods::nmethods_do(code_roots);
  }
  if (so & SO_AllCodeCache) {
    assert(code_roots != NULL, "must supply closure for code cache");

    // CMSCollector uses this to do intermediate-strength collections.
    // We scan the entire code cache, since CodeCache::do_unloading is not called.
    CodeCache::blobs_do(code_roots);
  }
  // Verify that the code cache contents are not subject to
  // movement by a scavenging collection.
  DEBUG_ONLY(CodeBlobToOopClosure
assert_code_is_non_scavengable(&assert_is_non_scavengable_closure,
!CodeBlobToOopClosure::FixRelocations));

DEBUG_ONLY(ScavengableNMethods::asserted_non_scavengable_nmethods_do(&assert_c
ode_is_non_scavengable));
}
```

Explaining ClassLoaderDataGraph::roots_cld_do(strong_cld_closure, weak_cld_closure);

```
void ClassLoaderDataGraph::roots_cld_do(CLDClosure* strong, CLDClosure* weak) {
  assert_locked_or_safepoint_weak(ClassLoaderDataGraph_lock);
  for (ClassLoaderData* cld = _head;  cld != NULL; cld = cld->_next) {
    CLDClosure* closure = cld->keep_alive() ? strong : weak;
    if (closure != NULL) {
      closure->do_cld(cld);
    }
  }
}
```

For defNewGeneration.cpp, the closure is CLDScanClosure cld_scan_closure(&scan_closure);

```
void CLDScanClosure::do_cld(ClassLoaderData* cld) {
  NOT_PRODUCT(ResourceMark rm);
  log_develop_trace(gc, scavenge)("CLDScanClosure::do_cld " PTR_FORMAT ", %s, dirty:
%s",
                                  p2i(cld),
                                  cld->loader_name_and_id(),
                                  cld->has_modified_oops() ? "true" : "false");

  // If the cld has not been dirtied we know that there's
  // no references into  the young gen and we can skip it.
  if (cld->has_modified_oops()) {

    // Tell the closure which CLD is being scanned so that it can be dirtied
    // if oops are left pointing into the young gen.
    _scavenge_closure->set_scanned_cld(cld);

    // Clean the cld since we're going to scavenge all the metadata.
    cld->oops_do(_scavenge_closure, ClassLoaderData::_claim_none,
/*clear_modified_oops*/true);

    _scavenge_closure->set_scanned_cld(NULL);
  }
}
```

Explaining Threads::oops_do(strong_roots, roots_from_code_p)

```
void Threads::oops_do(OopClosure* f, CodeBlobClosure* cf) {
  ALL_JAVA_THREADS(p) {
    p->oops_do(f, cf);
  }
  VMThread::vm_thread()->oops_do(f, cf);
}
```

```
#define ALL_JAVA_THREADS(X) \
  for (JavaThread* X : *ThreadsSMRSupport::get_java_thread_list())
```

```
void Thread::oops_do(OopClosure* f, CodeBlobClosure* cf) {
  // Record JavaThread to GC thread
  RememberProcessedThread rpt(this);
  oops_do_no_frames(f, cf);
  oops_do_frames(f, cf);
}
```

```
void Thread::oops_do_no_frames(OopClosure* f, CodeBlobClosure* cf) {
  // Do oop for ThreadShadow
  f->do_oop((oop*)&_pending_exception);
  handle_area()->oops_do(f);
}
```

```
void JavaThread::oops_do_frames(OopClosure* f, CodeBlobClosure* cf) {
  if (!has_last_Java_frame()) {
    return;
  }
  // Finish any pending lazy GC activity for the frames
  StackWatermarkSet::finish_processing(this, NULL /* context */, StackWatermarkKind::gc);
  // Traverse the execution stack
  for (StackFrameStream fst(this, true /* update */, false /* process_frames */); !fst.is_done();
fst.next()) {
    fst.current()->oops_do(f, cf, fst.register_map());
  }
}
```

```
  // Memory management
  void oops_do(OopClosure* f, CodeBlobClosure* cf, const RegisterMap* map) {
#if COMPILER2_OR_JVMCI
    DerivedPointerIterationMode dpim = DerivedPointerTable::is_active() ?
                                       DerivedPointerIterationMode::_with_table :
                                       DerivedPointerIterationMode::_ignore;
#else
    DerivedPointerIterationMode dpim = DerivedPointerIterationMode::_ignore;;
#endif
    oops_do_internal(f, cf, NULL, dpim, map, true);
  }
```

```
void frame::oops_do_internal(OopClosure* f, CodeBlobClosure* cf,
                             DerivedOopClosure* df, DerivedPointerIterationMode derived_mode,
                             const RegisterMap* map, bool use_interpreter_oop_map_cache) const {
#ifndef PRODUCT
  // simulate GC crash here to dump java thread in error report
  if (CrashGCForDumpingJavaThread) {
    char *t = NULL;
    *t = 'c';
  }
#endif
  if (is_interpreted_frame()) {
    oops_interpreted_do(f, map, use_interpreter_oop_map_cache);
  } else if (is_entry_frame()) {
    oops_entry_do(f, map);
  } else if (is_upcall_stub_frame()) {
    _cb->as_upcall_stub()->oops_do(f, *this);
  } else if (CodeCache::contains(pc())) {
    oops_code_blob_do(f, cf, df, derived_mode, map);
  } else {
    ShouldNotReachHere();
  }
}
```

```
void frame::oops_interpreted_do(OopClosure* f, const RegisterMap* map, bool
query_oop_map_cache) const {
  assert(is_interpreted_frame(), "Not an interpreted frame");
  Thread *thread = Thread::current();
  methodHandle m (thread, interpreter_frame_method());
  jint      bci = interpreter_frame_bci();

  assert(!Universe::heap()->is_in(m()),
         "must be valid oop");
  assert(m->is_method(), "checking frame value");
  assert((m->is_native() && bci == 0)  ||
        (!m->is_native() && bci >= 0 && bci < m->code_size()),
        "invalid bci value");

  // Handle the monitor elements in the activation
  for (
    BasicObjectLock* current = interpreter_frame_monitor_end();
    current < interpreter_frame_monitor_begin();
    current = next_monitor_in_interpreter_frame(current)
  ) {
#ifdef ASSERT
    interpreter_frame_verify_monitor(current);
#endif
    current->oops_do(f);
  }

  if (m->is_native()) {
    f->do_oop(interpreter_frame_temp_oop_addr());
  }

  // The method pointer in the frame might be the only path to the method's
  // klass, and the klass needs to be kept alive while executing. The GCs
  // don't trace through method pointers, so the mirror of the method's klass
  // is installed as a GC root.
  f->do_oop(interpreter_frame_mirror_addr());

  int max_locals = m->is_native() ? m->size_of_parameters() : m->max_locals();

  Symbol* signature = NULL;
  bool has_receiver = false;

  // Process a callee's arguments if we are at a call site
  // (i.e., if we are at an invoke bytecode)
  // This is used sometimes for calling into the VM, not for another
  // interpreted or compiled frame.
  if (!m->is_native()) {
    Bytecode_invoke call = Bytecode_invoke_check(m, bci);
```

```
    if (map != nullptr && call.is_valid()) {
      signature = call.signature();
      has_receiver = call.has_receiver();
      if (map->include_argument_oops() &&
          interpreter_frame_expression_stack_size() > 0) {
        ResourceMark rm(thread);  // is this right ???
        // we are at a call site & the expression stack is not empty
        // => process callee's arguments
        //
        // Note: The expression stack can be empty if an exception
        //       occurred during method resolution/execution. In all
        //       cases we empty the expression stack completely be-
        //       fore handling the exception (the exception handling
        //       code in the interpreter calls a blocking runtime
        //       routine which can cause this code to be executed).
        //       (was bug gri 7/27/98)
        oops_interpreted_arguments_do(signature, has_receiver, f);
      }
    }
  }

  InterpreterFrameClosure blk(this, max_locals, m->max_stack(), f);

  // process locals & expression stack
  InterpreterOopMap mask;
  if (query_oop_map_cache) {
    m->mask_for(bci, &mask);
  } else {
    OopMapCache::compute_one_oop_map(m, bci, &mask);
  }
  mask.iterate_oop(&blk);
}
```

```
void frame::oops_entry_do(OopClosure* f, const RegisterMap* map) const {
  assert(map != NULL, "map must be set");
  if (map->include_argument_oops()) {
    // must collect argument oops, as nobody else is doing it
    Thread *thread = Thread::current();
    methodHandle m (thread, entry_frame_call_wrapper()->callee_method());
    EntryFrameOopFinder finder(this, m->signature(), m->is_static());
    finder.arguments_do(f);
  }
  // Traverse the Handle Block saved in the entry frame
  entry_frame_call_wrapper()->oops_do(f);
}
```

```
void UpcallStub::oops_do(OopClosure* f, const frame& frame) {
  frame_data_for_frame(frame)->old_handles->oops_do(f);
}
```

```
void frame::oops_code_blob_do(OopClosure* f, CodeBlobClosure* cf, DerivedOopClosure*
df, DerivedPointerIterationMode derived_mode, const RegisterMap* reg_map) const {
  assert(_cb != NULL, "sanity check");
  assert((oop_map() == NULL) == (_cb->oop_maps() == NULL), "frame and _cb must agree
that oopmap is set or not");
  if (oop_map() != NULL) {
    if (df != NULL) {
      _oop_map->oops_do(this, reg_map, f, df);
    } else {
      _oop_map->oops_do(this, reg_map, f, derived_mode);
    }

    // Preserve potential arguments for a callee. We handle this by dispatching
    // on the codeblob. For c2i, we do
    if (reg_map->include_argument_oops()) {
      _cb->preserve_callee_argument_oops(*this, reg_map, f);
    }
  }
  // In cases where perm gen is collected, GC will want to mark
  // oops referenced from nmethods active on thread stacks so as to
  // prevent them from being collected. However, this visit should be
  // restricted to certain phases of the collection only. The
  // closure decides how it wants nmethods to be traced.
  if (cf != NULL)
    cf->do_code_blob(_cb);
}
```