This report highlights the methods involved to identify high latency, bottlenecks / hot paths in the application and the requisite steps taken to optimize the code.

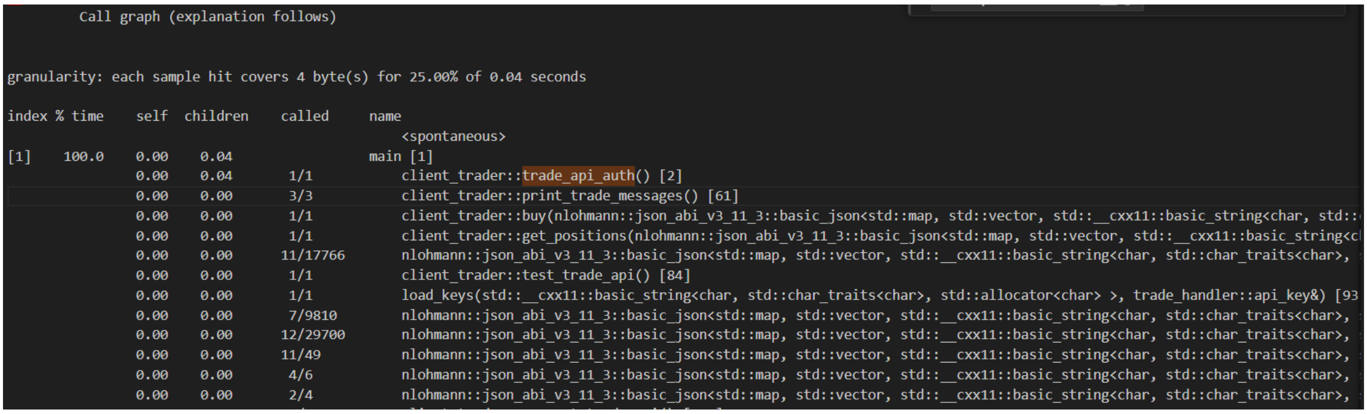
The code explanations are based on the code-review document. It is important to note that the code examples here refer to `class deribit` as a derived class from `class trade_handler`.`trade_handler` acts as an interface (abstract class).

1. CPU Optimization

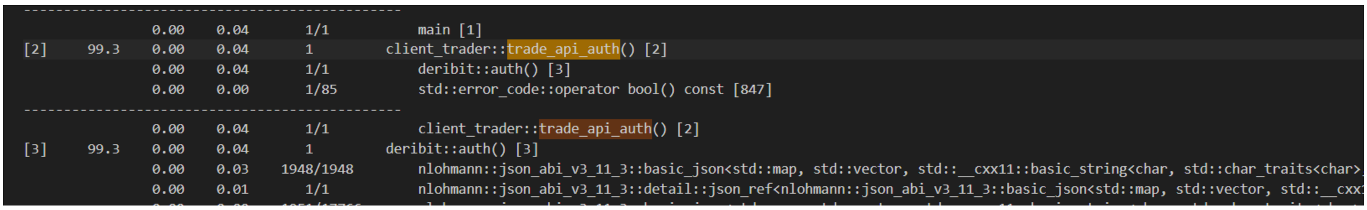
Identifying Initial Bottlenecks

Step 1

When profiling the application for the first time with `gprof` the following results were established. The bottleneck identified is `deribit::auth`. The function waits till the API returns an access token which is necessary for confirming authentication and may be required for private calls to the API. According to the following results, we must optimize the code which waits for the access token.



```
        Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 25.00% of 0.04 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.04                 main [1]
                0.00    0.04       1/1           client_trader::trade_api_auth() [2]
                0.00    0.00       3/3           client_trader::print_trade_messages() [61]
                0.00    0.00       1/1           client_trader::buy(nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::
                0.00    0.00       1/1           client_trader::get_positions(nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<c
                0.00    0.00   11/17766          nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00       1/1           client_trader::test_trade_api() [84]
                0.00    0.00       1/1           load_keys(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, trade_handler::api_key&) [93
                0.00    0.00    7/9810           nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00   12/29700          nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00     11/49           nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00       4/6           nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00       2/4           nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
```



```
-----------------------------------------------
                0.00    0.04       1/1           main [1]
[2]     99.3    0.00    0.04       1         client_trader::trade_api_auth() [2]
                0.00    0.04       1/1           deribit::auth() [3]
                0.00    0.00      1/85           std::error_code::operator bool() const [847]
-----------------------------------------------
                0.00    0.04       1/1           client_trader::trade_api_auth() [2]
[3]     99.3    0.00    0.04       1         deribit::auth() [3]
                0.00    0.03 1948/1948           nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>
                0.00    0.01       1/1           nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx
```

The line below refers to the function `json::parse()` from the nlohmann_json library.

```
0.00    0.00      13/15          nlohmann::json_abi_v3_11_3::basic_json<...>, ...
nlohmann::json_abi_v3_11_3::adl_serializer [1014]
```

The code for `deribit::auth()` before optimizations involved checking and parsing latest messages every iteration.

```
if(!result.ec) {
    using namespace std::chrono_literals;
```

```
    // wait for response
    json json_response;

    while(true) {
        auto msg = m_endpoint->get_latest_message(m_con_id);

        if(!msg || msg->length() <= WS_MSG_TYPE_LEN) continue;

        json_response = json::parse(msg->substr(WS_MSG_TYPE_LEN));

        if(json_response.contains("result") &&
json_response["result"].contains("access_token")) {
            m_access_token = json_response["result"]["access_token"];
            break;
        }
    }

    APP_LOG(log_flags::trade_handler, "(deribit) access token: " <<
m_access_token);
}
```

We can fix this by parsing the message only if a new message arrives. Using the statements

```
auto tmp = m_endpoint->get_latest_message(m_con_id);
if(tmp == msg) continue; // check if new message has arrived
```

Step 2

After fixing continuous parsing of messages as json, we profile the application again.

```
-----------------------------------------------
                0.00    0.04  420763/420763      deribit::auth() [3]
[4]     75.0    0.00    0.04  420763         websocket_endpoint::get_latest_message[abi:cxx11](int) [4]
                0.00    0.02  420775/420776      std::map<int, std::shared_ptr<connection_metadata>, std::less<int>, std::alloc
                0.01    0.00  420771/420773      std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
                0.00    0.00  420764/420769      std::map<int, std::shared_ptr<connection_metadata>, std::less<int>, std::alloc
                0.00    0.00  841562/841565      std::_Rb_tree_const_iterator<std::pair<int const, std::shared_ptr<connection_m
```

The function performs slow due to large number of calls to `get_latest_message`. We can reduce the calls to `get_latest_message` by reducing the polling, by adding a thread delay.

```
auto thread_sleep_time = 100ms;

while(true) {
    // add a thread sleep to avoid excessively polling get_latest_message
    std::this_thread::sleep_for(thread_sleep_time);
```

We have successfully reduced the number of calls to `get_latest_message` from ~420,000 to 3 calls. Additionally the time spent on the function `deribit::auth()` reduced from 0.04s to 0.00s (below `gprof`'s measurement threshold).

```
-----------------------------------------------
                0.00    0.00     1/1           client_trader::trade_api_auth() [4280]
[4760]    0.0   0.00    0.00     1         deribit::auth() [4760]
                0.00    0.00    13/15          nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00    10/47          nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00     8/25          nlohmann::json_abi_v3_11_3::basic_json<std::map, std::vector, std::__cxx11::basic_string<char, std::char_traits<char>,
                0.00    0.00     3/3           websocket_endpoint::get_latest_message[abi:cxx11](int) [2615]
```

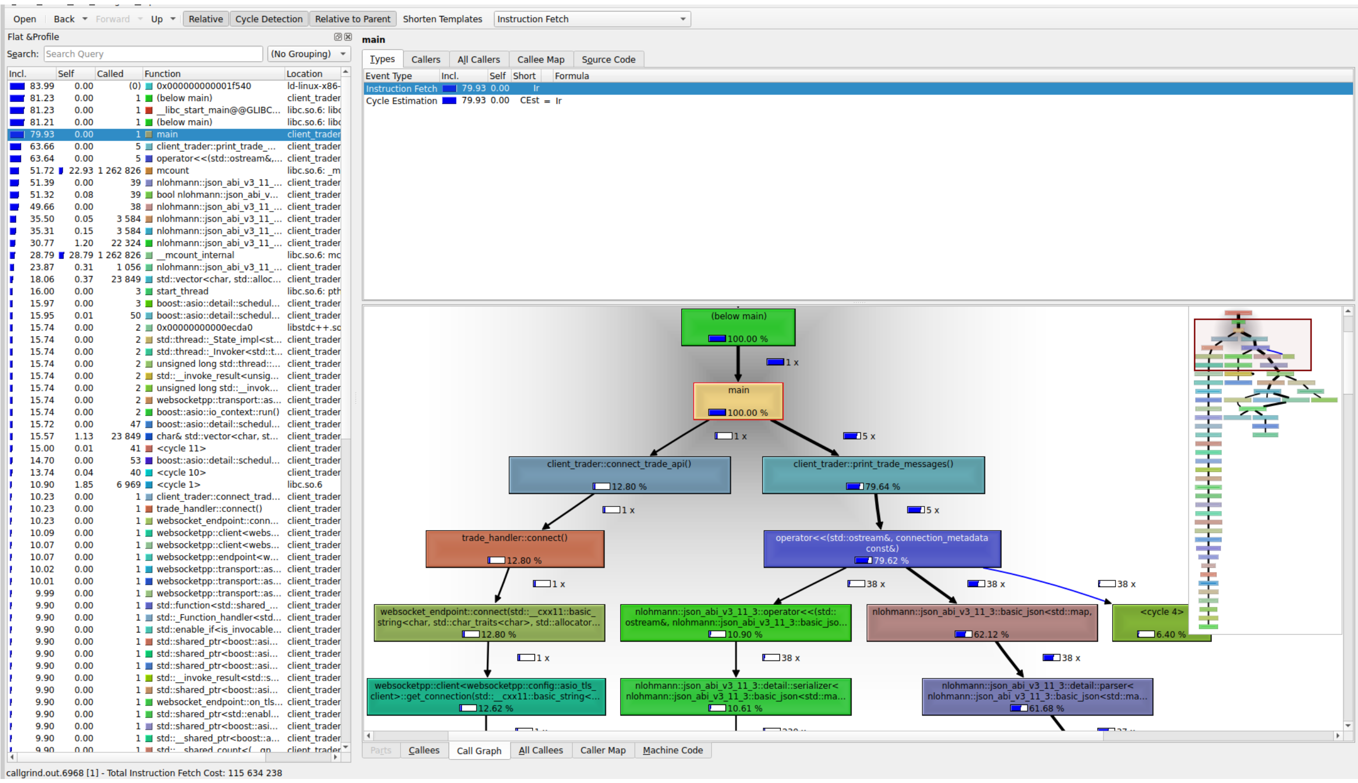
Additionally the final implementation of `deribit::auth()` involves a timeout which exits the function if the access token is not retrieved within a fixed amount of time.

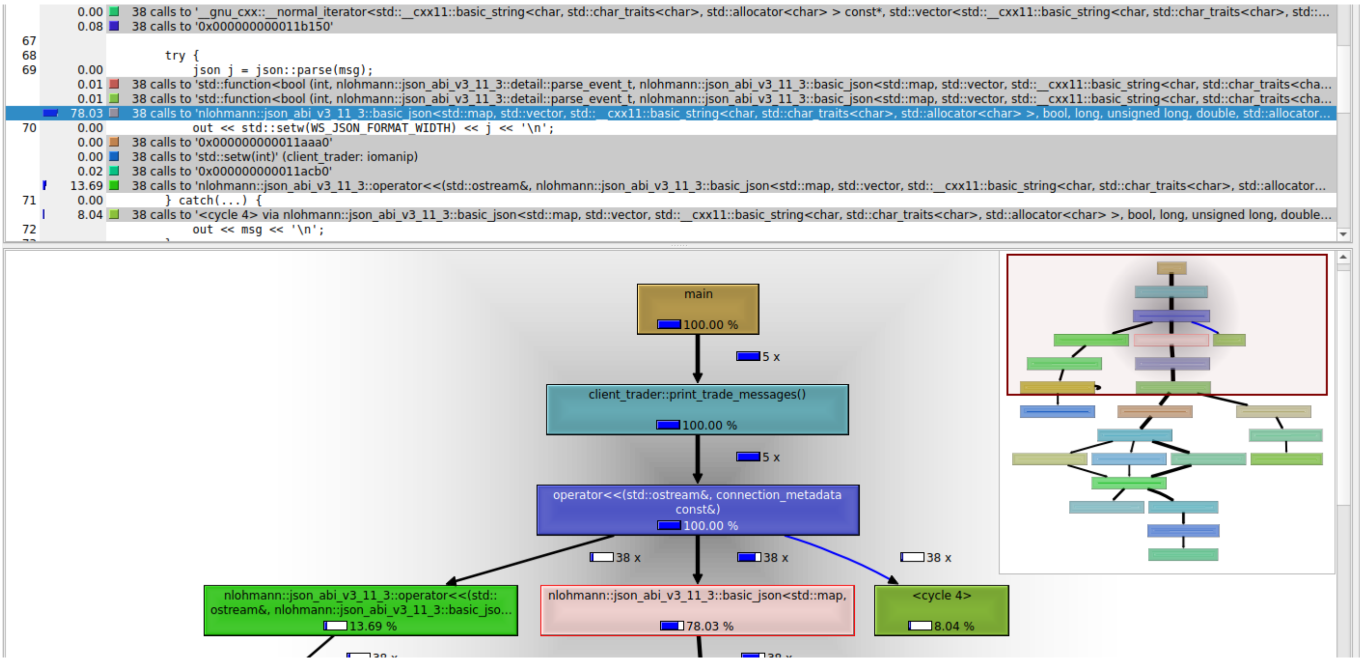
The final function definition is available in `api/deribit.h`.

Profiling with callgrind

Since `gprof` does not give high frequency sampling, we will use the `callgrind` tool and analyze it using KCachegrind.

Step 1





The profile shows a graph of the function calls and the percentage of time on each call. According to the first image, we see `callgrind` recognizes `print_trade_messages` as a major bottleneck with 79% of the time spent on it. The second image highlights that majority of the time is spent in "pretty-printing" the json messages using nlohmann json library. This method is invoked when we run the command `deribit_show`.

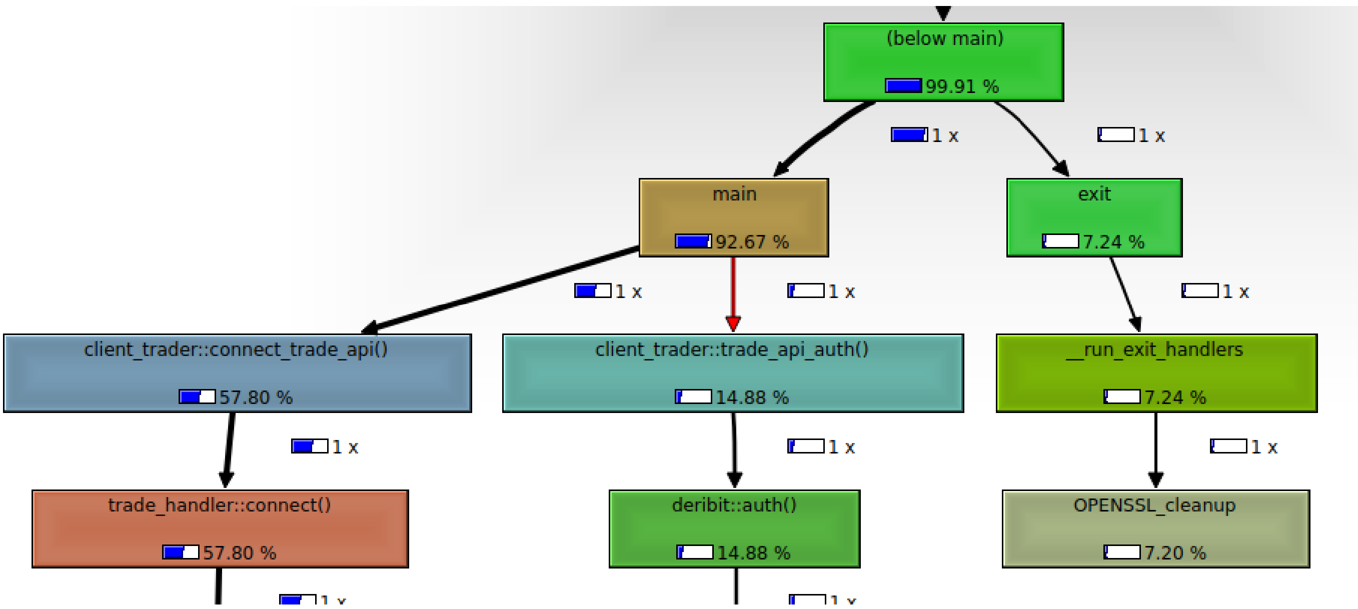
The lines of code responsible are shown in the second image.

Step 2

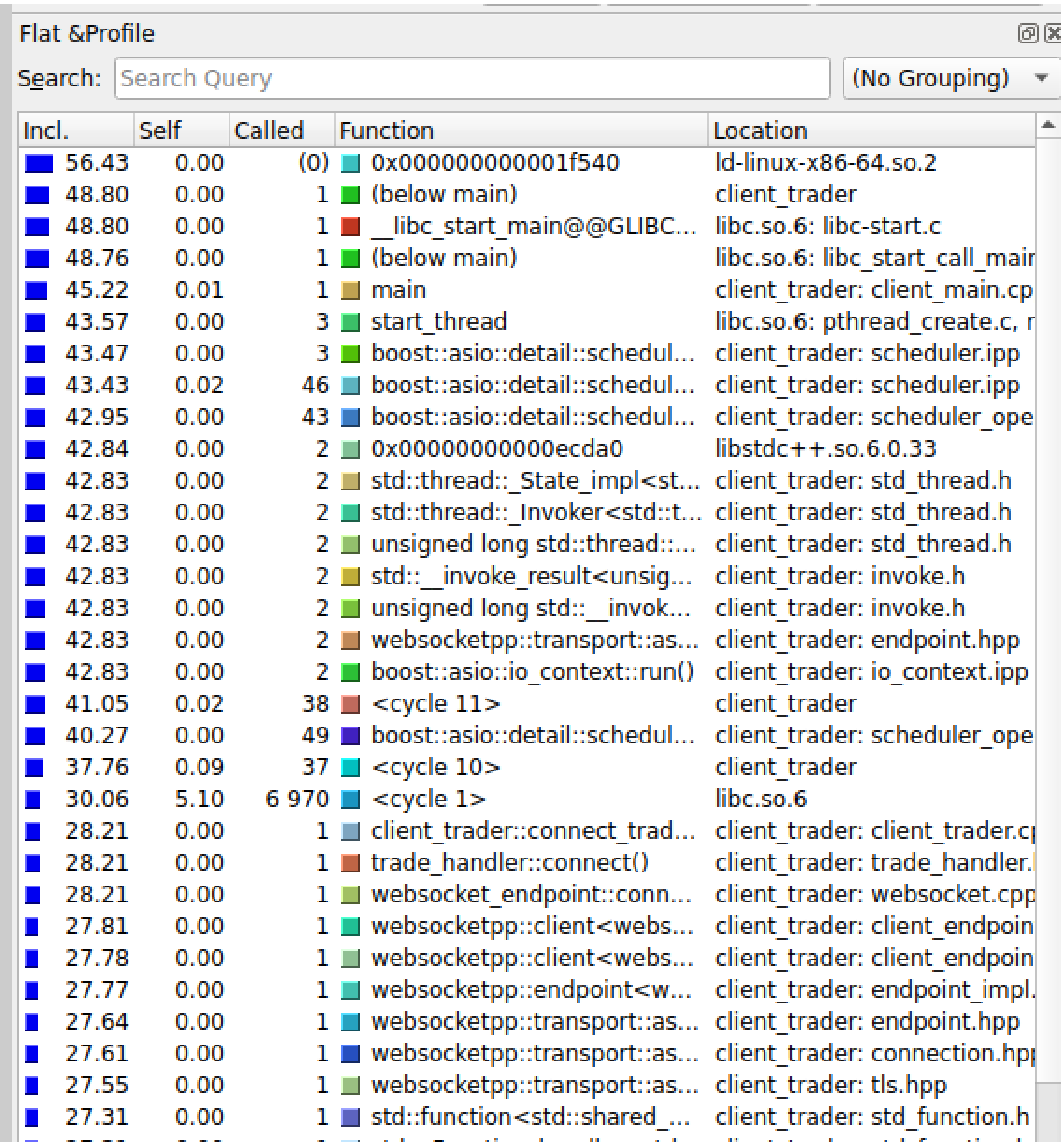
To identify other major hot paths, we should temporarily disable pretty printing of json messages in the network communication.

Now, we rerun the application and invoke all the possible deribit commands in our application. The below graph and show that the trading methods such as `deribit_buy`, `deribit_positions` etc. have a much lower execution time than the connect and auth methods.

Call graph:



Flat profile:



Incl.	Self	Called	Function	Location
56.43	0.00	(0)	0x000000000001f540	ld-linux-x86-64.so.2
48.80	0.00	1	(below main)	client_trader
48.80	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
48.76	0.00	1	(below main)	libc.so.6: libc_start_call_mair
45.22	0.01	1	main	client_trader: client_main.cp
43.57	0.00	3	start_thread	libc.so.6: pthread_create.c, r
43.47	0.00	3	boost::asio::detail::schedul...	client_trader: scheduler.ipp
43.43	0.02	46	boost::asio::detail::schedul...	client_trader: scheduler.ipp
42.95	0.00	43	boost::asio::detail::schedul...	client_trader: scheduler_ope
42.84	0.00	2	0x00000000000ecda0	libstdc++.so.6.0.33
42.83	0.00	2	std::thread::_State_impl<st...	client_trader: std_thread.h
42.83	0.00	2	std::thread::_Invoker<std::t...	client_trader: std_thread.h
42.83	0.00	2	unsigned long std::thread::...	client_trader: std_thread.h
42.83	0.00	2	std::__invoke_result<unsig...	client_trader: invoke.h
42.83	0.00	2	unsigned long std::__invok...	client_trader: invoke.h
42.83	0.00	2	websocketpp::transport::as...	client_trader: endpoint.hpp
42.83	0.00	2	boost::asio::io_context::run()	client_trader: io_context.ipp
41.05	0.02	38	<cycle 11>	client_trader
40.27	0.00	49	boost::asio::detail::schedul...	client_trader: scheduler_ope
37.76	0.09	37	<cycle 10>	client_trader
30.06	5.10	6 970	<cycle 1>	libc.so.6
28.21	0.00	1	client_trader::connect_trad...	client_trader: client_trader.c
28.21	0.00	1	trade_handler::connect()	client_trader: trade_handler.
28.21	0.00	1	websocket_endpoint::conn...	client_trader: websocket.cpp
27.81	0.00	1	websocketpp::client<webs...	client_trader: client_endpoin
27.78	0.00	1	websocketpp::client<webs...	client_trader: client_endpoin
27.77	0.00	1	websocketpp::endpoint<w...	client_trader: endpoint_impl.
27.64	0.00	1	websocketpp::transport::as...	client_trader: endpoint.hpp
27.61	0.00	1	websocketpp::transport::as...	client_trader: connection.hp
27.55	0.00	1	websocketpp::transport::as...	client_trader: tls.hpp
27.31	0.00	1	std::function<std::shared_...	client_trader: std_function.h

- We have already optimized the `deribit::auth`, however it has a relatively higher execution time since it waits for the API to return the access token.
- Our `client_trader::connect` method directly calls the underlying connect method from `websocketpp` library.

Since, these methods are only invoked only once on application start a relatively higher execution time may be acceptable according to requirements.

Analysis of buy function

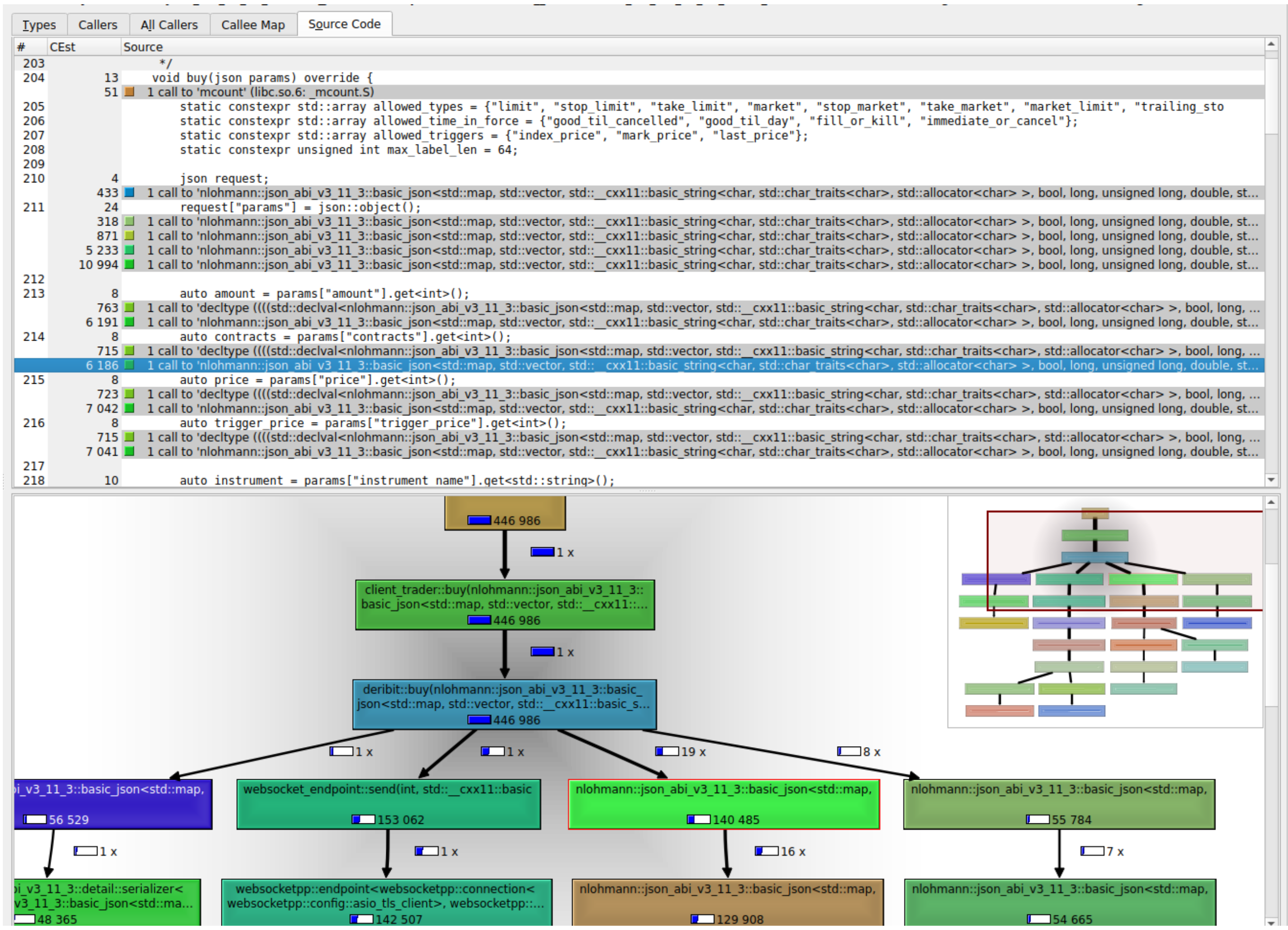
We will focus on the `deribit::buy` function since it sends the largest request from all of our deribit commands, making it suitable for analysis.

We can confirm that requests with large request body such as `private/buy` do not cause bottlenecks and work at low latency. the `self` time for `deribit::buy` is much lower compared to the auth and connect methods. The highlighted blue function in the below image refers to the deribit buy function.

Note: the self and include times mentioned are in relative mode, hence they refer to timing relative to other methods mentioned.

Incl.	Self	Called	Function	Location
10 429 542	5	1	SSL_CTX_new	libssl.so.3
10 429 537	201	1	SSL_CTX_new_ex	libssl.so.3
10 383 997	4 584 909	218 329	mcount	libc.so.6: _mcount.
9 364 878	2 784	32	boost::asio::ssl::detail::engi...	client_trader: engi
9 031 789	659	5	boost::asio::ssl::detail::io_o...	client_trader: io.hp
8 942 293	80	4	boost::asio::ssl::detail::han...	client_trader: hand
8 942 021	124	4	boost::asio::ssl::detail::engi...	client_trader: engi
8 933 435	56	4	boost::asio::ssl::detail::engi...	client_trader: engi
8 933 187	8	4	0x000000000011b1f0	(unknown)
8 933 179	23	4	SSL_connect	libssl.so.3
8 933 041	1 346	4	SSL_do_handshake	libssl.so.3
8 360 635	25 852	199	0x00000000001fa4f0	libcrypto.so.3
8 347 324	7 334	193	0x00000000001faaa0	libcrypto.so.3
8 122 217	128	2	boost::asio::detail::complet...	client_trader: com
8 101 217	46	2	void boost::asio::detail::ha...	client_trader: hand
8 100 957	36	2	void boost_asio_handler_in...	client_trader: hand
8 100 591	30	2	void boost::asio::detail::asi...	client_trader: wrap
8 100 463	38	2	void boost_asio_handler_in...	client_trader: hand
8 100 095	38	2	void boost::asio::asio_hand...	client_trader: hand
8 099 959	26	2	boost::asio::detail::rewrapp...	client_trader: wrap
8 099 835	56	2	boost::asio::detail::binder2...	client_trader: bind
7 891 790	456	15	boost::asio::detail::epoll_re...	client_trader: epoll
7 481 009	1 716	7	0x0000000000070c80	libssl.so.3
7 365 395	633	1	0x000000000002f790	libssl.so.3
6 342 827	660	33	EVP_CIPHER_fetch	libcrypto.so.3
6 317 970	782	23	0x000000000003c3b0	libssl.so.3
6 302 291	46	23	0x000000000486ecf0	(unknown)
6 069 730	3 584	64	0x000000000021f660	libcrypto.so.3
5 961 937	2 880	64	0x000000000021f260	libcrypto.so.3
5 959 057	7 552	64	0x000000000022ed40	libcrypto.so.3
5 814 985	7 954	64	0x000000000021f080	libcrypto.so.3
5 799 088	5 799 088	218 329	__mcount_internal	libc.so.6: mcount.c
5 724 397	10 212	246	0x000000000021f480	libcrypto.so.3
5 545 775	640	10	boost::asio::detail::complet...	client_trader: com
5 451 578	230	10	void boost::asio::detail::ha...	client_trader: hand
5 450 310	180	10	void boost_asio_handler_in...	client_trader: hand
5 448 512	150	10	void boost::asio::detail::asi...	client_trader: wrap
5 447 888	190	10	void boost_asio_handler_in...	client_trader: hand
5 446 080	190	10	void boost::asio::asio_hand...	client_trader: hand
5 445 416	130	10	boost::asio::detail::rewrapp...	client_trader: wrap
5 444 812	280	10	boost::asio::detail::binder2...	client_trader: bind
5 239 435	60	15	ASN1_item_d2i	libcrypto.so.3
5 239 375	705	15	ASN1_item_d2i_ex	libcrypto.so.3
5 238 670	2 778	15	0x00000000000df6c0	libcrypto.so.3
5 225 335	2 483	17	0x00000000000e0900	libcrypto.so.3
5 219 815	86 637	20	<cycle 8>	libcrypto.so.3
5 207 491	6	3	0x000000000486d280	(unknown)
5 207 485	63	3	d2i_X509	libcrypto.so.3
4 404 331	663 939	11 346	0x000000000021d450	libcrypto.so.3
4 054 171	387	3	0x000000000038bfc0 <cy...	libcrypto.so.3
4 041 510	189	3	OSSL_DECODER_CTX_new_...	libcrypto.so.3
3 624 638	158 402	3 705	0x000000000021f780	libcrypto.so.3
3 426 308	7 659	207	0x00000000001f9f00	libcrypto.so.3
446 986	427	1	deribit::buy(nlohmann::jso...	client_trader: derib

Step 1



The methods `nlohmann::json_abi_...::basic_json` refer directly or indirectly to the conversion of JSON data to C++ data types / strings.

The `trade_handler` buy method is defined to take a single object as the set of parameters which the function definition will access.

- This also ensures that a different API can use seperate parameters as the arguments passed to the function.

The original code used a common json object for parameters:

```
void trade_handler::buy(json params);
```

Original code:

```
void deribit::buy(json params) {
    json request;

    auto instrument = params["instrument_name"].get<std::string>();

    if(instrument.empty()) {
        // verify parameters ...
```

```
    }

    request["params"] = json::object();
    request["params"]["instrument_name"] = instrument;
}
```

Since, the conversion of data types using `.get()` is costly, we can optimize the performance by using a common struct for the parameters

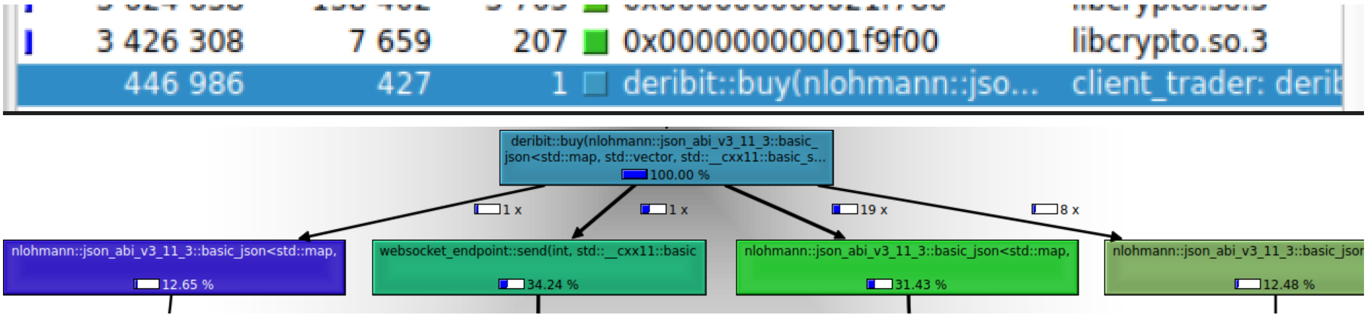
```
struct buy_params {
    float amount;
    float contracts;
    float price;
    float trigger_price;

    std::string instrument;
    std::string type;
    std::string label;
    std::string time_in_force;
    std::string trigger;

    // for edit and cancel orders
    std::string order_id
};
```

The optimized code below does not require any conversion of data types while not losing the abstraction layer of `trade_handler`.

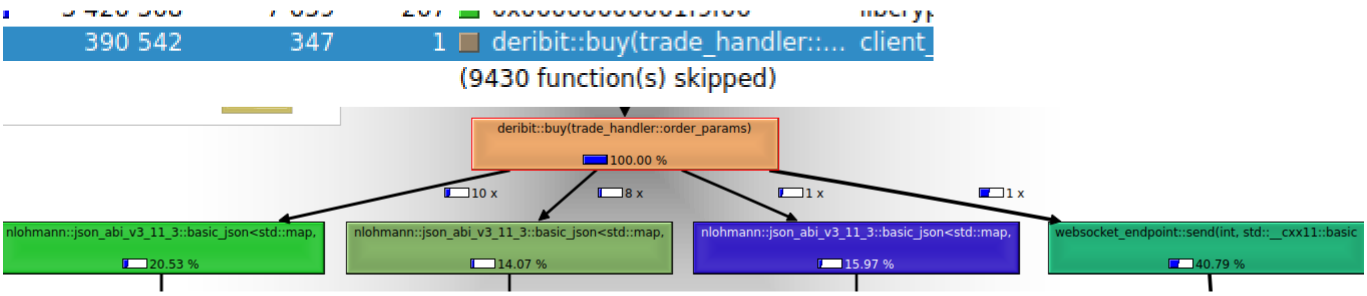
```
void deribit::buy(trade_handler::buy_params params) {
    if(params.instrument.empty()) {
        // verify parameters ...
    }

    json request;
    request["params"] = json::object();
    request["params"]["instrument_name"] = params.instrument;
}
```

Result

Before:

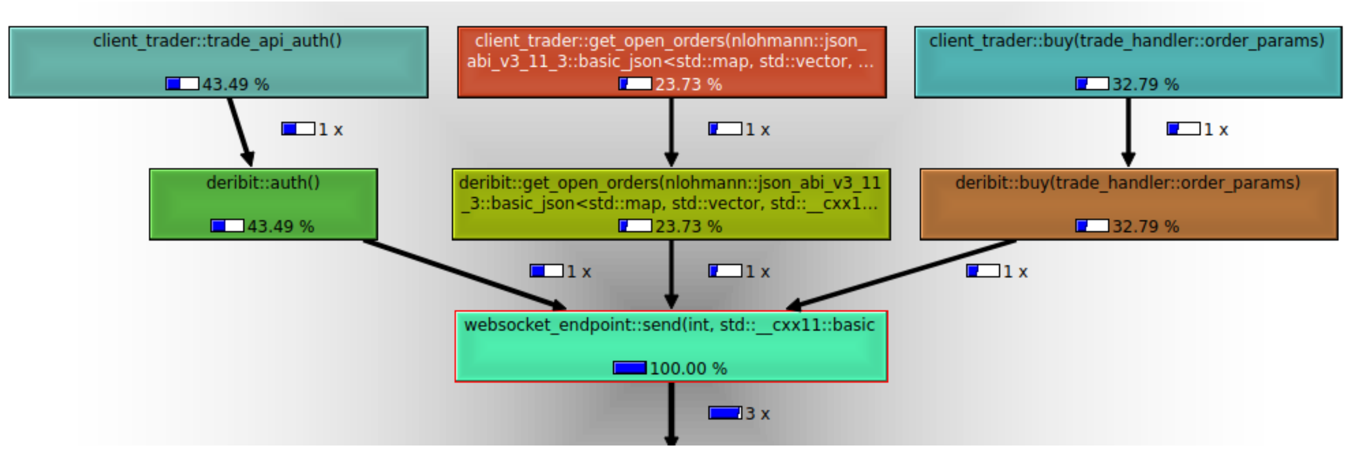


After:



Analysis of send requests

The following graph indicates the amount of time spent on send requests from the application. As we expect `deribit::buy` takes a longer amount of time than requests such as `deribit::get_open_orders`. The detailed analysis of timings can be seen in the benchmark section.



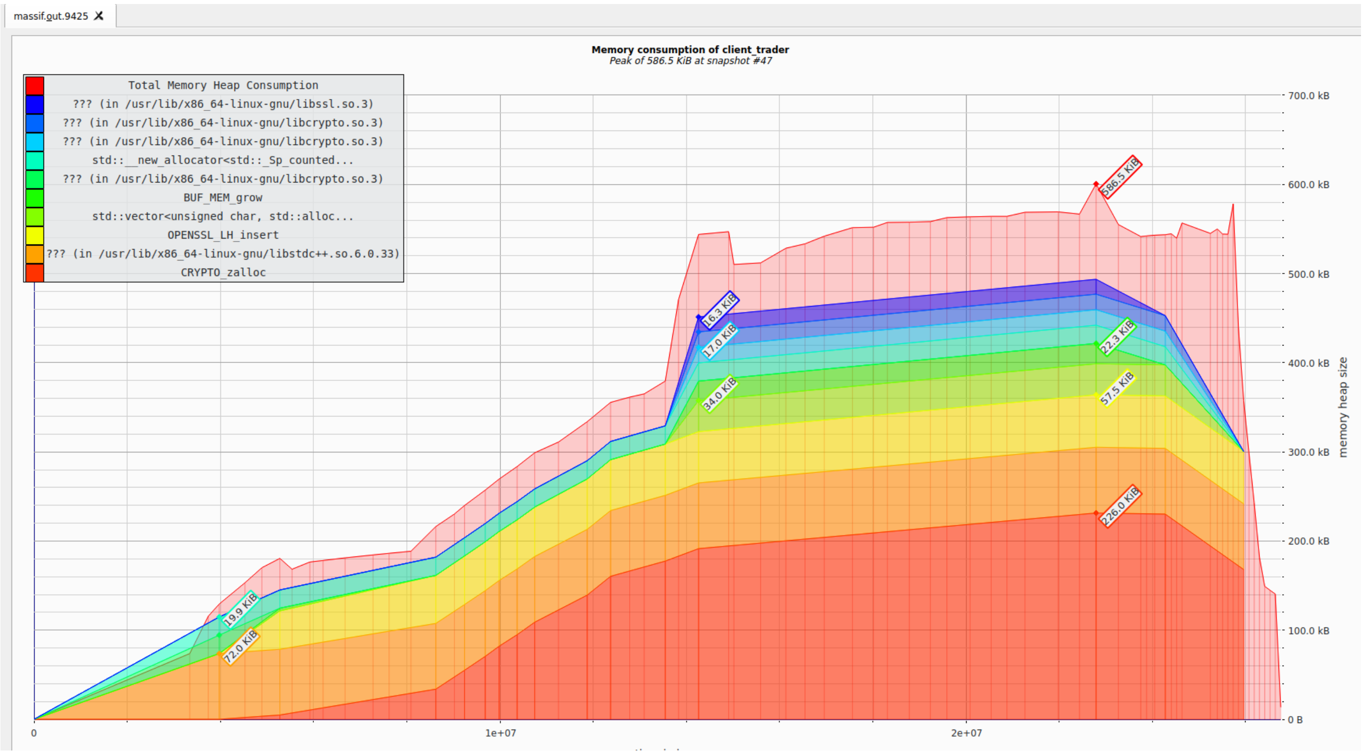
2. Memory Optimization

Memory Profiling

First Run

The heap memory profiling was done using the tool `massif` and visualized using massif-visualizer program.

```
valgrind --tool=massif client_trader
```



The peak heap memory usage is 586kb. The snapshot of peak usage is as follows:

586.5 KiB: Snapshot #47 (peak)
- 226.0 KiB: CRYPTO_zalloc (in /usr/lib/x86_64-linux-gnu/libcr...
  - 63.6 KiB: in 119 places, all below massif's threshold (1....
  - 47.9 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
  - 31.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
  - 27.9 KiB: OPENSSL_LH_new (in /usr/lib/x86_64-linux-gnu..
  - 17.3 KiB: CRYPTO_THREAD_lock_new (in /usr/lib/x86_64-..
  - 15.9 KiB: OPENSSL_LH_new (in /usr/lib/x86_64-linux-gnu..
  - 7.8 KiB: OPENSSL_sk_new_reserve (in /usr/lib/x86_64-lin..
  - 7.4 KiB: SSL_new (in /usr/lib/x86_64-linux-gnu/libssl.so.3)
  - 7.2 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
- 72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0...
- 59.4 KiB: in 178 places, all below massif's threshold (1.00%)
- 57.5 KiB: OPENSSL_LH_insert (in /usr/lib/x86_64-linux-gnu/li..
- 34.0 KiB: std::__new_allocator<unsigned char>::allocate(u...
- 22.3 KiB: BUF_MEM_grow (in /usr/lib/x86_64-linux-gnu/libcr...
- 19.9 KiB: std::__new_allocator<std::_Sp_counted_ptr_inplac..
- 17.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
- 17.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
- 16.3 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libssl.so.3)
- 16.1 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libssl.so.3)
- 11.0 KiB: OPENSSL_LH_insert (in /usr/lib/x86_64-linux-gnu/li..
- 9.6 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libcrypto.so.3)
- 8.3 KiB: CRYPTO_strndup (in /usr/lib/x86_64-linux-gnu/libcr...

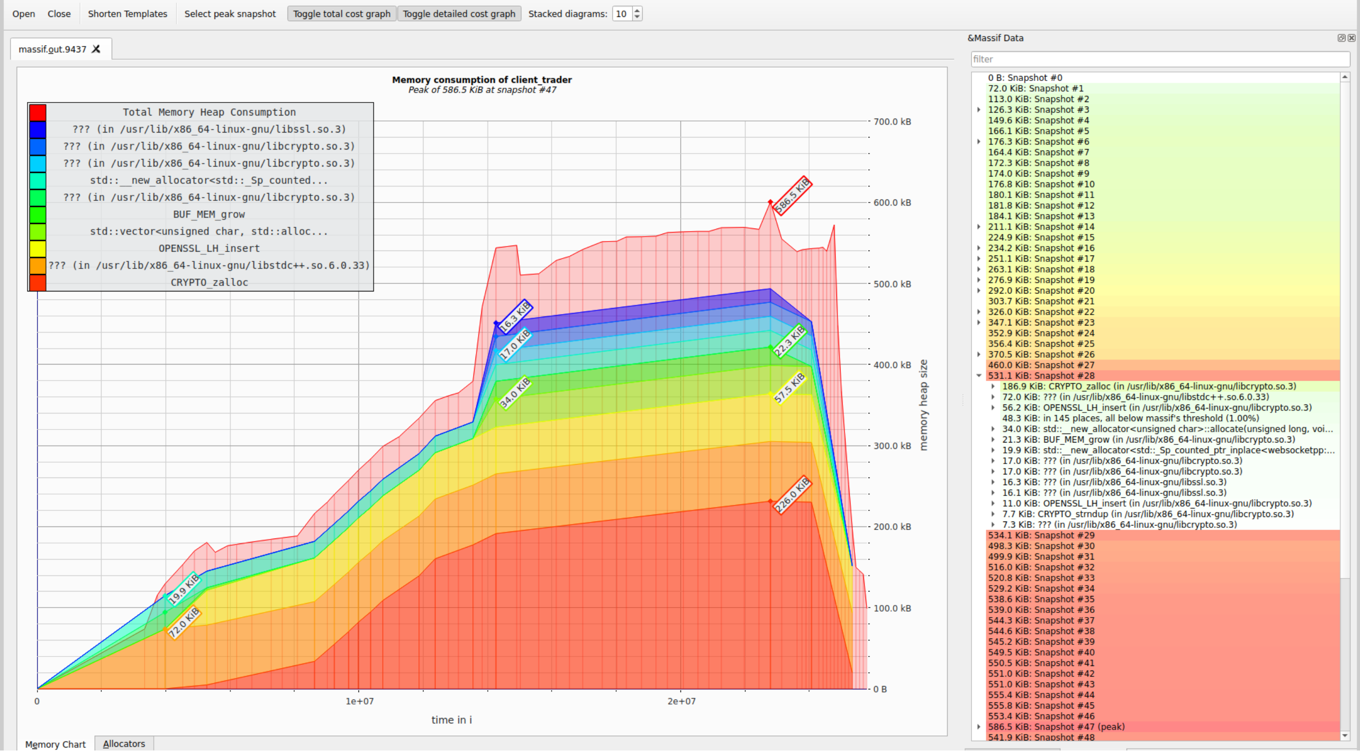
The allocators used were:

massif.out.9425

Function	Peak	Location
CRYPTO_zalloc	226.0 KiB	in /usr/lib/
???	72.0 KiB	in /usr/lib/
below threshold	59.7 KiB	
OPENSSL_LH_insert	57.5 KiB	in /usr/lib/
std::__new_allocator<unsigned char>::allocate(unsigned long, void const*)	34.0 KiB	new_alloca
BUF_MEM_grow	22.3 KiB	in /usr/lib/
std::__new_allocator<std::_Sp_counted_ptr_inplace<websocketpp::connection<websocketpp::config::asio_tls_client>, std::allocator<void>, (__gnu_cxx::_Lock_policy)2> >::allocate(unsigned long, void const*)	19.9 KiB	new_alloca
???	17.0 KiB	in /usr/lib/
???	16.3 KiB	in /usr/lib/
CRYPTO_strndup	8.3 KiB	in /usr/lib/
_IO_file_doallocate	6.0 KiB	filedoalloc.
OBJ_NAME_add	5.3 KiB	in /usr/lib/
boost::asio::execution_context::service* boost::asio::detail::service_registry::create<boost::asio::detail::strand_service, boost::asio::io_context>(void*)	1.6 KiB	service_re

This shows us that we did not do any significant heap allocation and majority of it was done by the `websocketspp` library as shown in the graph legend.

Second Run



A second profile of the application showed a similar graph for heap memory allocation.

Analysis of Memory Leaks

We can detect memory leaks in the application using `valgrind`.

```
valgrind --leak-check=full -v ./client_trader
```

```
==9504== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==9504==
==9504== 1 errors in context 1 of 2:
==9504== Mismatched new/delete size value: 120
==9504==    at 0x484A5B9: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9504==    by 0x11DF40: main (client_main.cpp:250)
==9504==  Address 0x53e5200 is 0 bytes inside a block of size 152 alloc'd
==9504==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9504==    by 0x11C800: main (client_main.cpp:44)
==9504==
==9504== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

These memory leak errors refer to the lines:

```
// client_main.cpp
44:    trade_handler* deribit_handler = new deribit{};
250:   delete deribit_handler;
```

This is a subtle memory leak caused in our application due to varying size of base and derived classes. Since `trade_handler` and `deribit_handler` have a different size we cannot use the base class pointer to `delete` the allocation.

This can be fixed by managing the derived class's memory using a smart pointer. Later we extract the raw pointer and interface it through `trade_handler*`.

```
std::unique_ptr<deribit> deribit_uptr = std::make_unique<deribit>();
trade_handler* deribit_handler = deribit_uptr.get();
```

Result:

```
==10118== HEAP SUMMARY:
==10118==     in use at exit: 0 bytes in 0 blocks
==10118==   total heap usage: 14,360 allocs, 14,360 frees, 1,903,315 bytes
allocated
==10118==
==10118== All heap blocks were freed -- no leaks are possible
==10118==
==10118== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

However, we should still define a virtual destructor for `trade_handler` as we may use raw pointer using `new` to instantiate a `deribit` type object.

```
virtual ~trade_handler() {}
```

Reference: https://stackoverflow.com/questions/461203/when-to-use-virtual-destructors

3. Code Optimization Methods

Thread Management

The application uses 2 threads, one main thread and when for the running the websocket endpoint.

```
// run endpoint on seperate thread
m_thread = websocketpp::lib::make_shared<websocketpp::lib::thread>(&client::run,
&m_endpoint);
```

- This ensures that other computations or user interaction can occur while network requests are processed in the background

The thread is gracefully joined to the main thread when our `websocket_endpoint` class (`websocket/websocket.cpp`) goes out of scope. The thread is joined when all connections are closed in the destructor.

```
websocket_endpoint::~websocket_endpoint() {
    // close connections
    for (con_list::const_iterator it = m_connection_list.begin(); it !=
m_connection_list.end(); ++it) {
        // ...
    m_endpoint.close(it->second->get_hdl(),
websocketpp::close::status::going_away, "", ec);
    }

    // wait till thread is complete
    m_thread->join();
}
```

Network Optimizations

- The application uses TLS configuration for secure communication with the server
- A map data structure is used to query a list of active connections quickly.

```
typedef std::map<con_id_type, connection_metadata::ptr> con_list;
con_list m_connection_list;
```

- The code for `websocket_endpoint` is exception safe and returns error codes instead, making it perform at low latency and avoid uncaught exceptions / memory leaks.
- The configuration of websocketspp client enables concurrency by default. We can additionally enable `permessage_deflate` for large requests.

```
typedef websocketpp::client<websocketpp::config::asio_tls_client> client;
```

4. Latency Benchmarking

The latency benchmarking is done using a special class defined for benchmarking.

```
class benchmark {
public:
    benchmark(std::string lab): label{lab} {}
```

```
    void reset(std::string lab = "");

    void start();
    void end();
private:
    std::string label;

    bool started = false;
    std::chrono::time_point<std::chrono::high_resolution_clock> start_time;
    std::chrono::time_point<std::chrono::high_resolution_clock> end_time;
};
```

In order to support end to end benchmarking (as well as across threads), we have defined a `benchmark g_benchmark {"g_benchmark"};` in `client_main.cpp` which can be accessed by other source files using `extern`.

For example:

```
// client_main.cpp
g_benchmark.reset("e2e_" + order_type + "_order_" + "benchmark");
g_benchmark.start();

if(order_type == "buy")
    trader.buy(params);
```

```
// websocket.cpp
websocket_endpoint::send_result websocket_endpoint::send(con_id_type id,
std::string message) {
    // benchmark send request
    benchmark send_benchmark {"send_request_benchmark"};
    send_benchmark.start();

    // ...

    // end send request benchmark
    send_benchmark.end();
    // end global benchmark
    g_benchmark.end();

    return result;
}
```

This gives the following benchmarking results.

```
Created connection with id 0
Enter Command: deribit_auth
```

```
[      7792ms] Started benchmark: e2e_auth_benchmark
[      7792ms] Started benchmark: send_request_benchmark
[      7792ms] Benchmark: send_request_benchmark, took 69 us
[      7792ms] Benchmark: e2e_auth_benchmark, took 131 us
[      8001ms] trade_handler: (deribit) access token: ...
Enter Command: deribit_buy
Enter buy order details
Instrument: BTC-PERPETUAL
Amount: 50
Contracts:
Price: 50
Type:
Label:
Time in force:
Trigger:
Trigger Price:
[     23847ms] Started benchmark: e2e_buy_order_benchmark
[     23847ms] trade_handler: (deribit) Buy order request sent. Check details
[     23847ms] Started benchmark: send_request_benchmark
[     23847ms] Benchmark: send_request_benchmark, took 48 us
[     23847ms] Benchmark: e2e_buy_order_benchmark, took 220 us
Enter Command: deribit_show
```

`e2e_buy_order_benchmark` refers to the end-to-end trading latency, while `send_request_benchmark` only refers to the time taken to propagate the message.

Additional benchmarking showed that send requests were always < `100us`, and end-to-end buy trading loop was on the order of ~`150-200us`.