Profiling Using the Intel Performance Profilers

Ryan Honeyager

November 21, 2019

JEDI Topic Discussion Meeting

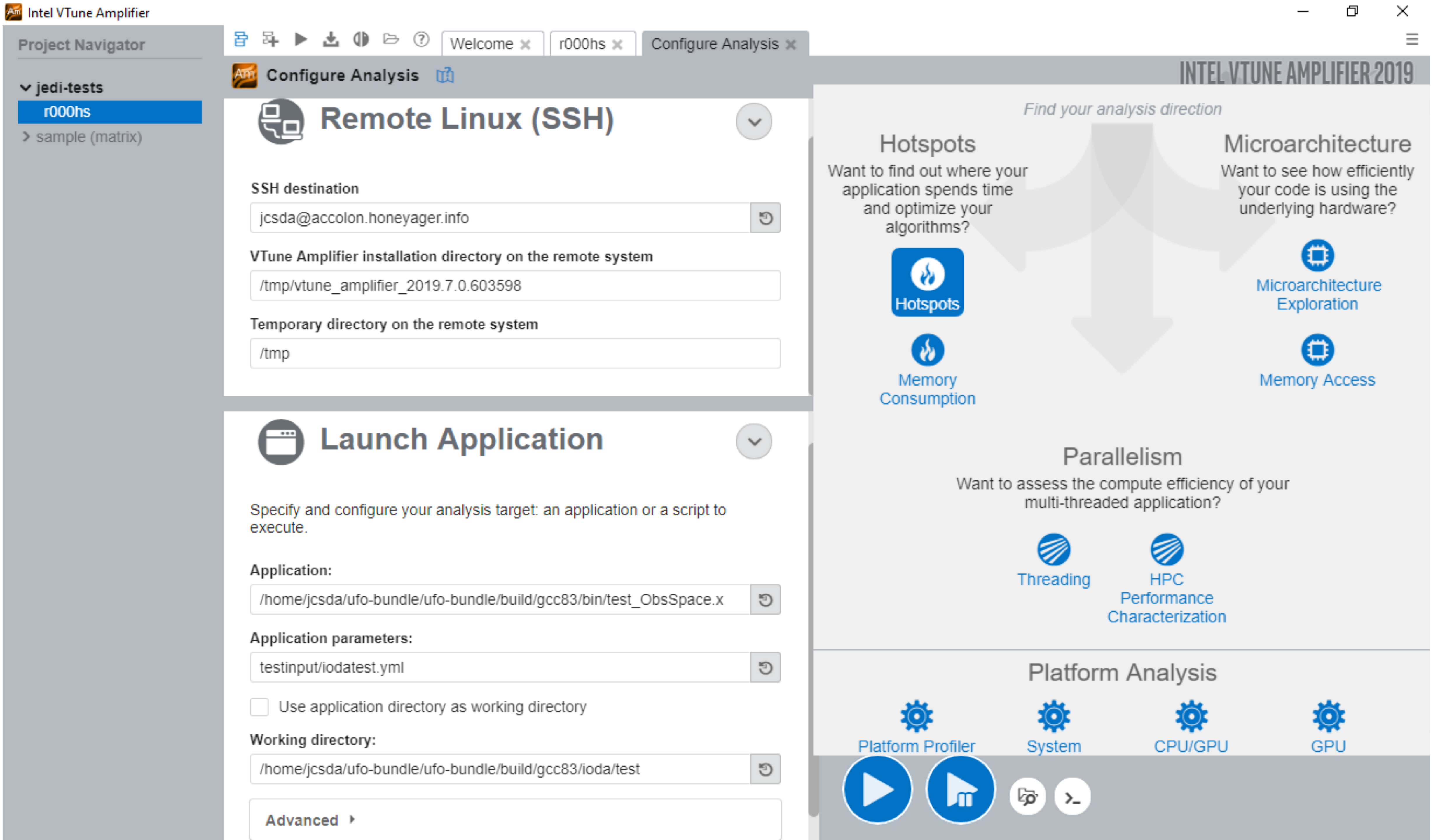
Intel-Provided Tools

- Intel Advisor – Helps optimize programs to use vectorization and shared-memory threading.
- Intel Inspector – A memory and thread checking and debugging tool.
- Intel VTune Amplifier (Profiler) – Performs many kinds of code profiling.
  - Only discussing VTune Amplifier today

- All tools are free and can be downloaded from Intel's website.
- They all work with C++ and Fortran, support MPI, and work with GCC, Intel compilers, Clang. No special compiler options needed beyond building in Debug and RelWithDebInfo modes.
- All are installed on Hera already (module load vtune inspector advisor).

VTune Amplifier (soon to be renamed VTune Profiler)

- Performs many kinds of code profiling:
  - Examine code hotspots by CPU utilization
  - Threading / MPI efficiency
  - Memory consumption
- Can profile using either CPU instructions (mostly Intel processors) or with software emulation. Defaults to polling every 10 ms.
- Profiling cost varies – hotspot analysis is <5-10%, memory consumption analysis is 2-5x.
- Has both GUI and console interfaces. Supports remote profiling via SSH, and can also save / load profiling results for future analysis.



Remote Linux (SSH)

SSH destination: jcsda@accolon.honeyager.info

VTune Amplifier installation directory on the remote system: /tmp/vtune_amplifier_2019.7.0.603598

Temporary directory on the remote system: /tmp

Launch Application

Specify and configure your analysis target: an application or a script to execute.

Application: /home/jcsda/ufo-bundle/ufo-bundle/build/gcc83/bin/test_ObsSpace.x

Application parameters: testinput/iodatest.yml

☐ Use application directory as working directory

Working directory: /home/jcsda/ufo-bundle/ufo-bundle/build/gcc83/ioda/test

Hotspots

Want to find out where your application spends time and optimize your algorithms?

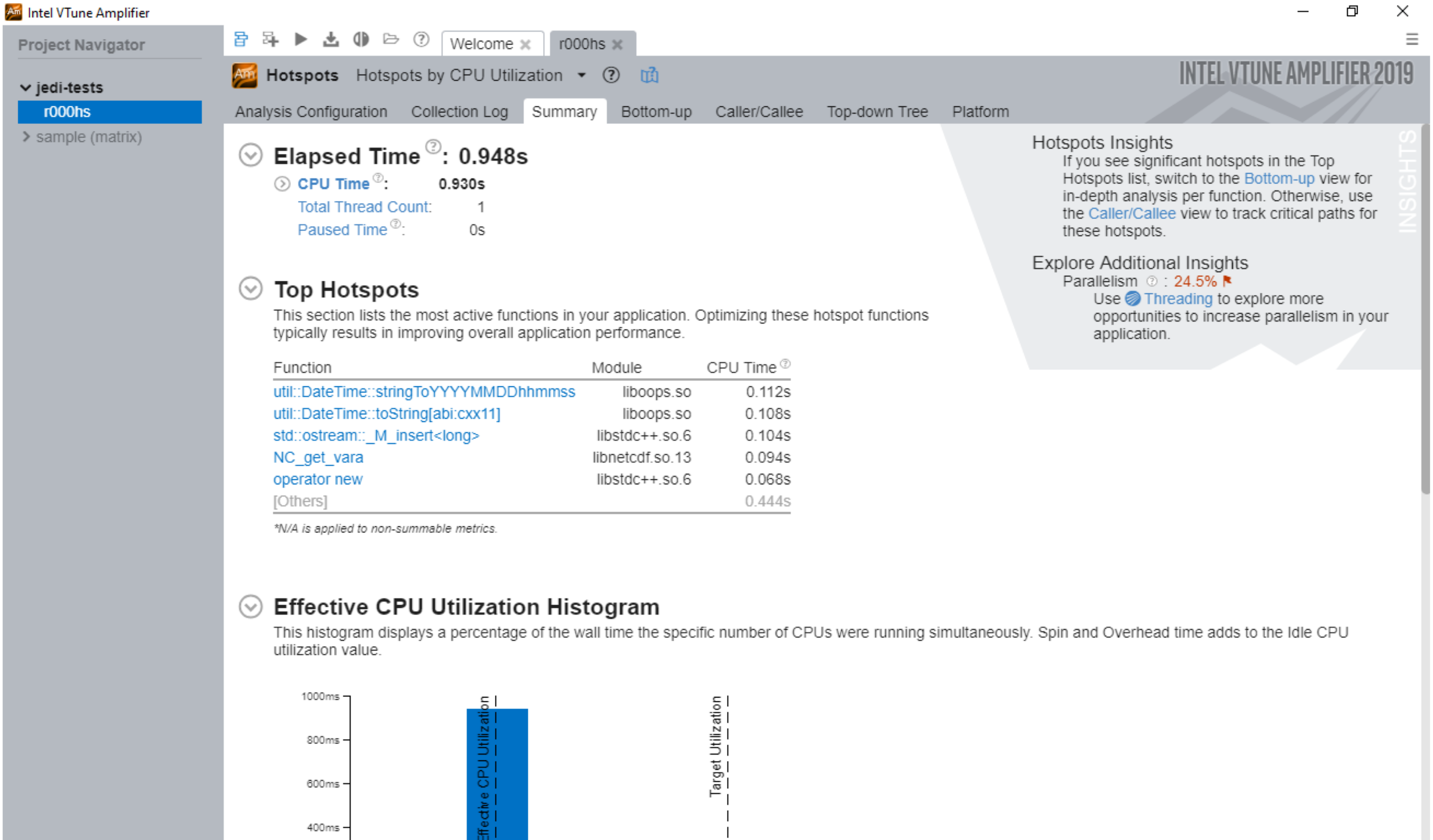
Microarchitecture

Want to see how efficiently your code is using the underlying hardware?

Parallelism

Want to assess the compute efficiency of your multi-threaded application?

Platform Analysis



Elapsed Time: 0.948s

CPU Time: 0.930s
Total Thread Count: 1
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
util::DateTime::stringToYYYYMMDDhhmmss	liboops.so	0.112s
util::DateTime::toString[abi:cxx11]	liboops.so	0.108s
std::ostream::_M_insert<long>	libstdc++.so.6	0.104s
NC_get_vara	libnetcdf.so.13	0.094s
operator new	libstdc++.so.6	0.068s
[Others]		0.444s

*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

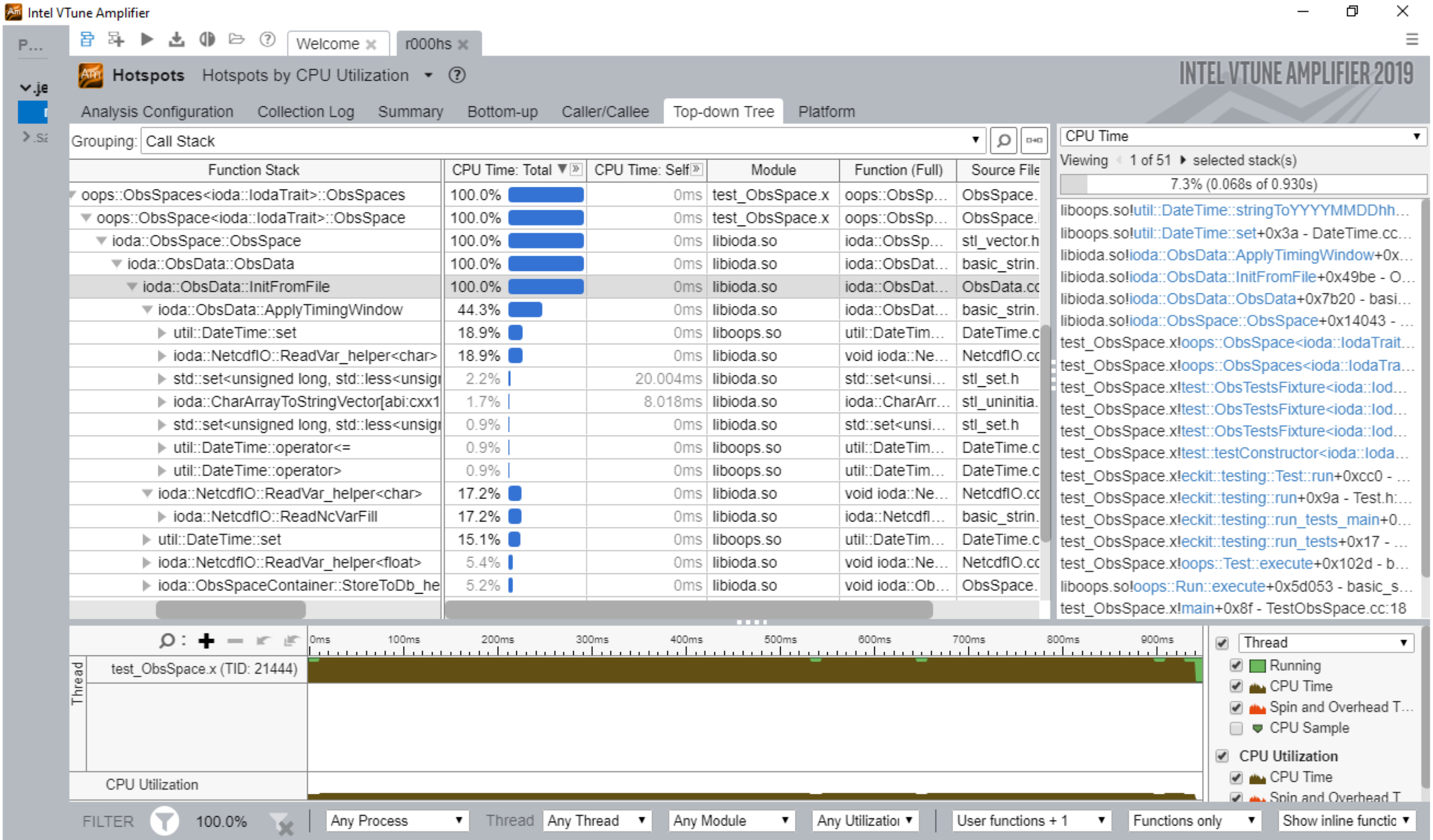
Explore Additional Insights

Parallelism : 24.5%

Use Threading to explore more opportunities to increase parallelism in your application.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Grouping: Call Stack

Function Stack	CPU Time: Total	CPU Time: Self	Module	Function (Full)	Source File
oops::ObsSpaces<ioda::IodaTrait>::ObsSpaces	100.0%	0ms	test_ObsSpace.x	oops::ObsSp…	ObsSpace.
oops::ObsSpace<ioda::IodaTrait>::ObsSpace	100.0%	0ms	test_ObsSpace.x	oops::ObsSp…	ObsSpace.
ioda::ObsSpace::ObsSpace	100.0%	0ms	libioda.so	ioda::ObsSp…	stl_vector.h
ioda::ObsData::ObsData	100.0%	0ms	libioda.so	ioda::ObsDat…	basic_strin.
ioda::ObsData::InitFromFile	100.0%	0ms	libioda.so	ioda::ObsDat…	ObsData.cc
ioda::ObsData::ApplyTimingWindow	44.3%	0ms	libioda.so	ioda::ObsDat…	basic_strin.
util::DateTime::set	18.9%	0ms	liboops.so	util::DateTim…	DateTime.c
ioda::NetcdfIO::ReadVar_helper<char>	18.9%	0ms	libioda.so	void ioda::Ne…	NetcdfIO.cc
std::set<unsigned long, std::less<unsigi	2.2%	20.004ms	libioda.so	std::set<unsi…	stl_set.h
ioda::CharArrayToStringVector[abi:cxx1	1.7%	8.018ms	libioda.so	ioda::CharArr…	stl_uninitia.
std::set<unsigned long, std::less<unsigi	0.9%	0ms	libioda.so	std::set<unsi…	stl_set.h
util::DateTime::operator<=	0.9%	0ms	liboops.so	util::DateTim…	DateTime.c
util::DateTime::operator>	0.9%	0ms	liboops.so	util::DateTim…	DateTime.c
ioda::NetcdfIO::ReadVar_helper<char>	17.2%	0ms	libioda.so	void ioda::Ne…	NetcdfIO.cc
ioda::NetcdfIO::ReadNcVarFill	17.2%	0ms	libioda.so	ioda::Netcdfl…	basic_strin.
util::DateTime::set	15.1%	0ms	liboops.so	util::DateTim…	DateTime.c
ioda::NetcdfIO::ReadVar_helper<float>	5.4%	0ms	libioda.so	void ioda::Ne…	NetcdfIO.cc
ioda::ObsSpaceContainer::StoreToDb_he	5.2%	0ms	libioda.so	void ioda::Ob…	ObsSpace.

CPU Time

Viewing 1 of 51 selected stack(s)

7.3% (0.068s of 0.930s)

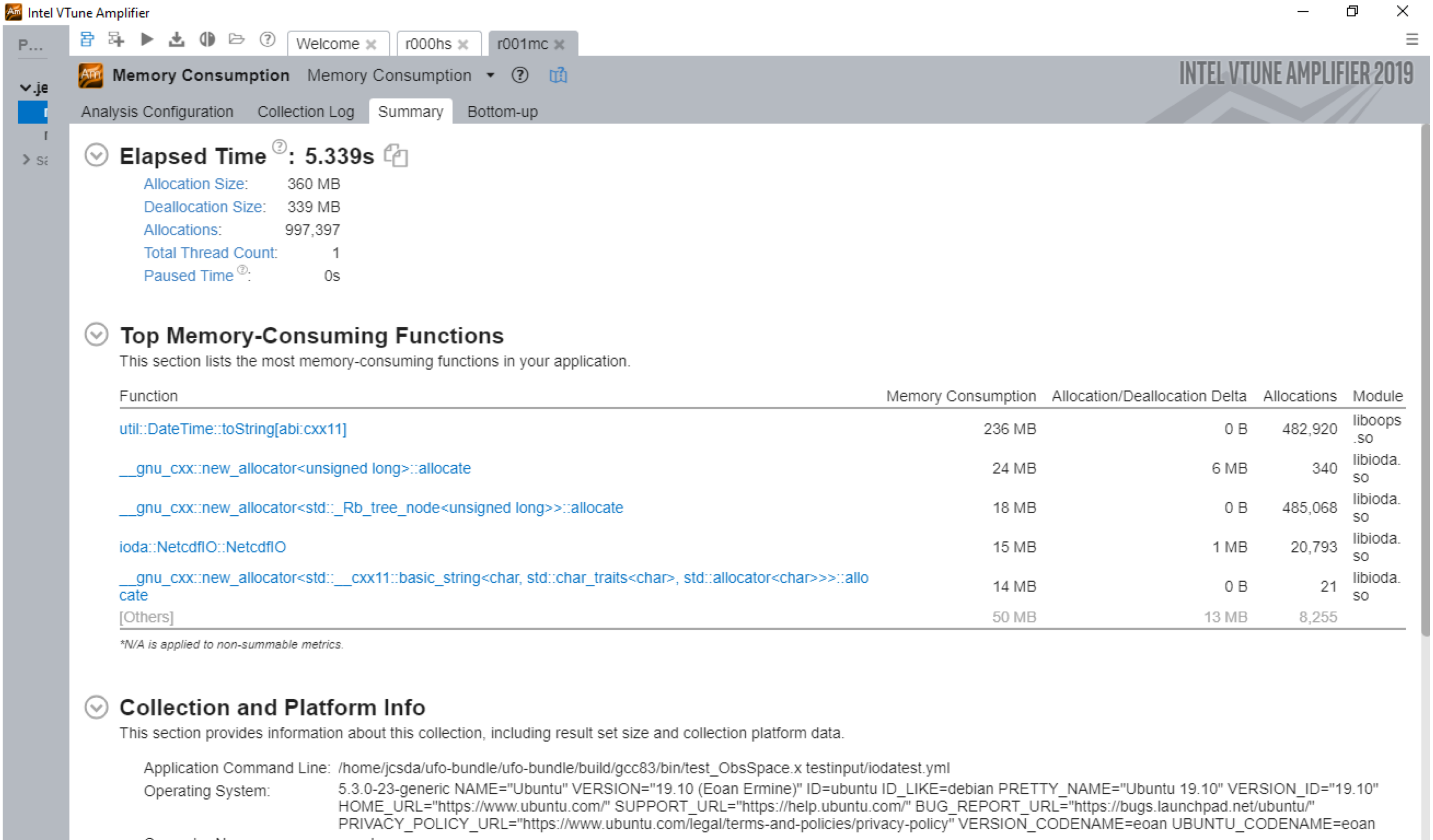
```
liboops.so!util::DateTime::stringToYYYYMMDDhh…
liboops.so!util::DateTime::set+0x3a - DateTime.cc…
libioda.so!ioda::ObsData::ApplyTimingWindow+0x…
libioda.so!ioda::ObsData::InitFromFile+0x49be - O…
libioda.so!ioda::ObsData::ObsData+0x7b20 - basi…
libioda.so!ioda::ObsSpace::ObsSpace+0x14043 - …
test_ObsSpace.x!oops::ObsSpace<ioda::IodaTrait…
test_ObsSpace.x!oops::ObsSpaces<ioda::IodaTra…
test_ObsSpace.x!test::ObsTestsFixture<ioda::Iod…
test_ObsSpace.x!test::ObsTestsFixture<ioda::Iod…
test_ObsSpace.x!test::ObsTestsFixture<ioda::Iod…
test_ObsSpace.x!test::testConstructor<ioda::Ioda…
test_ObsSpace.x!eckit::testing::Test::run+0xcc0 - …
test_ObsSpace.x!eckit::testing::run+0x9a - Test.h:…
test_ObsSpace.x!eckit::testing::run_tests_main+0…
test_ObsSpace.x!eckit::testing::run_tests+0x17 - …
test_ObsSpace.x!oops::Test::execute+0x102d - b…
liboops.so!oops::Run::execute+0x5d053 - basic_s…
test_ObsSpace.x!main+0x8f - TestObsSpace.cc:18
```

test_ObsSpace.x (TID: 21444)

CPU Utilization

☑ Thread
☑ 🟩 Running
☑ CPU Time
☑ Spin and Overhead T…
□ CPU Sample
☑ CPU Utilization
☑ CPU Time
☑ Spin and Overhead T

FILTER 100.0% Any Process Thread Any Thread Any Module Any Utilizatio User functions + 1 Functions only Show inline functic



Elapsed Time: 5.339s

Allocation Size:	360 MB
Deallocation Size:	339 MB
Allocations:	997,397
Total Thread Count:	1
Paused Time:	0s

Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

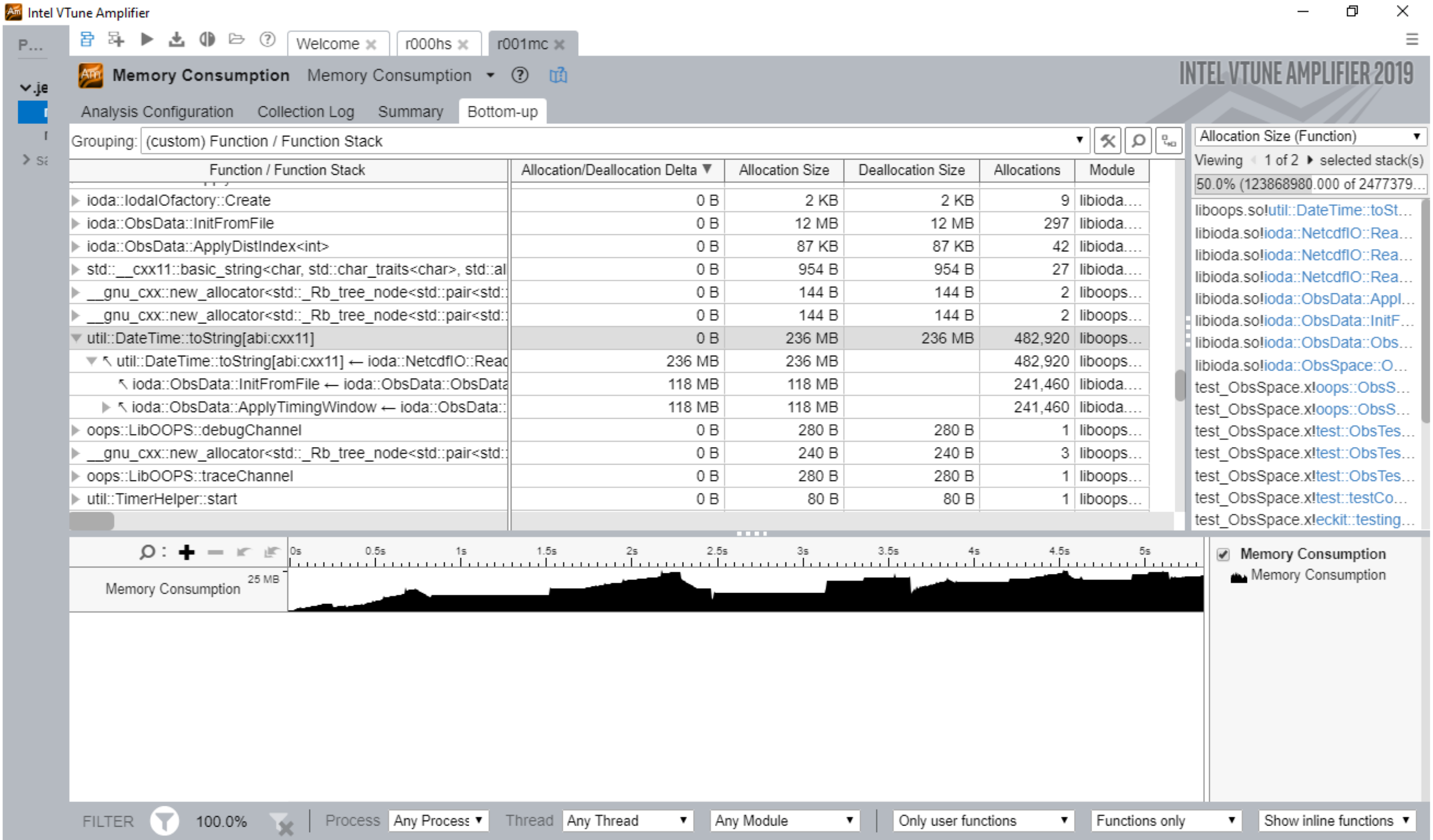
Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
util::DateTime::toString[abi:cxx11]	236 MB	0 B	482,920	liboops.so
__gnu_cxx::new_allocator<unsigned long>::allocate	24 MB	6 MB	340	libioda.so
__gnu_cxx::new_allocator<std::_Rb_tree_node<unsigned long>>::allocate	18 MB	0 B	485,068	libioda.so
ioda::NetcdfIO::NetcdfIO	15 MB	1 MB	20,793	libioda.so
__gnu_cxx::new_allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>::allocate	14 MB	0 B	21	libioda.so
[Others]	50 MB	13 MB	8,255	

N/A is applied to non-summable metrics.

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line:	/home/jcsda/ufo-bundle/ufo-bundle/build/gcc83/bin/test_ObsSpace.x testinput/iodatest.yml
Operating System:	5.3.0-23-generic NAME="Ubuntu" VERSION="19.10 (Eoan Ermine)" ID=ubuntu ID_LIKE=debian PRETTY_NAME="Ubuntu 19.10" VERSION_ID="19.10" HOME_URL="https://www.ubuntu.com/" SUPPORT_URL="https://help.ubuntu.com/" BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/" PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy" VERSION_CODENAME=eoan UBUNTU_CODENAME=eoan



Memory Consumption Memory Consumption

Grouping: (custom) Function / Function Stack

Function / Function Stack	Allocation/Deallocation Delta ▼	Allocation Size	Deallocation Size	Allocations	Module
▶ ioda::IodaIOfactory::Create	0 B	2 KB	2 KB	9	libioda....
▶ ioda::ObsData::InitFromFile	0 B	12 MB	12 MB	297	libioda....
▶ ioda::ObsData::ApplyDistIndex<int>	0 B	87 KB	87 KB	42	libioda....
▶ std::__cxx11::basic_string<char, std::char_traits<char>, std::al	0 B	954 B	954 B	27	libioda....
▶ __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::	0 B	144 B	144 B	2	liboops...
▶ __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::	0 B	144 B	144 B	2	liboops...
▼ util::DateTime::toString[abi:cxx11]	0 B	236 MB	236 MB	482,920	liboops...
▼ ↖ util::DateTime::toString[abi:cxx11] ← ioda::NetcdfIO::Read	236 MB	236 MB		482,920	liboops...
↖ ioda::ObsData::InitFromFile ← ioda::ObsData::ObsData	118 MB	118 MB		241,460	libioda....
▶ ↖ ioda::ObsData::ApplyTimingWindow ← ioda::ObsData::	118 MB	118 MB		241,460	libioda....
▶ oops::LibOOPS::debugChannel	0 B	280 B	280 B	1	liboops...
▶ __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::	0 B	240 B	240 B	3	liboops...
▶ oops::LibOOPS::traceChannel	0 B	280 B	280 B	1	liboops...
▶ util::TimerHelper::start	0 B	80 B	80 B	1	liboops...

Allocation Size (Function)

Viewing ◀ 1 of 2 ▶ selected stack(s)

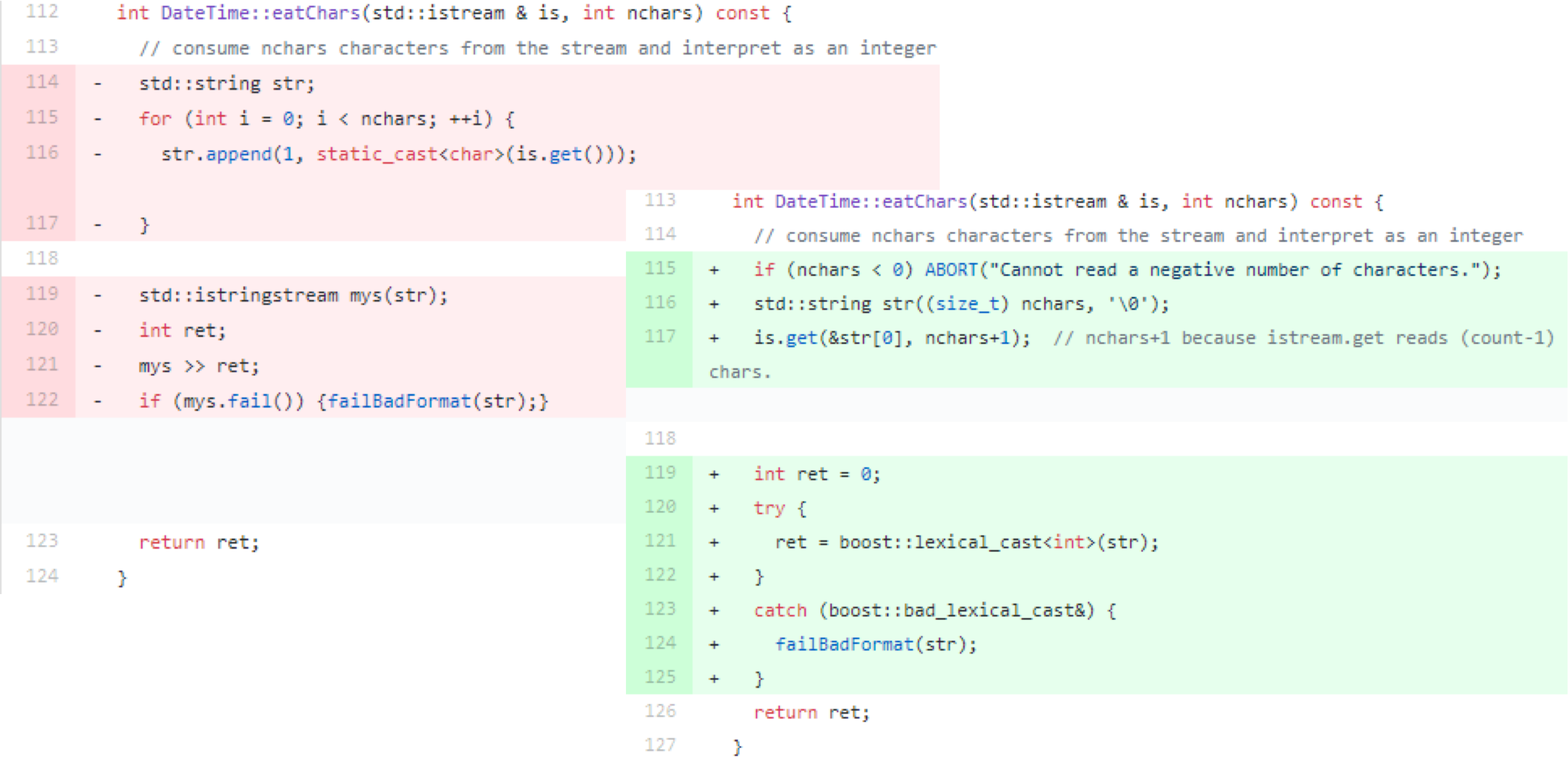
50.0% (123868980.000 of 2477379...

```
liboops.so!util::DateTime::toSt...
libioda.so!ioda::NetcdfIO::Rea...
libioda.so!ioda::NetcdfIO::Rea...
libioda.so!ioda::NetcdfIO::Rea...
libioda.so!ioda::ObsData::Appl...
libioda.so!ioda::ObsData::InitF...
libioda.so!ioda::ObsData::Obs...
libioda.so!ioda::ObsSpace::O...
test_ObsSpace.x!oops::ObsS...
test_ObsSpace.x!oops::ObsS...
test_ObsSpace.x!test::ObsTes...
test_ObsSpace.x!test::ObsTes...
test_ObsSpace.x!test::ObsTes...
test_ObsSpace.x!test::testCo...
test_ObsSpace.x!eckit::testing...
```

☑ Memory Consumption
Memory Consumption

FILTER 100.0% Process Any Process Thread Any Thread Any Module Only user functions Functions only Show inline functions

Example: See https://github.com/JCSDA/oops/pull/442. Reduced execution time of test by 45% (1.68 to 0.93 seconds) by rewriting ten lines of code.



```
   int DateTime::eatChars(std::istream & is, int nchars) const {
     // consume nchars characters from the stream and interpret as an integer
-    std::string str;
-    for (int i = 0; i < nchars; ++i) {
-      str.append(1, static_cast<char>(is.get()));
-    }

-    std::istringstream mys(str);
-    int ret;
-    mys >> ret;
-    if (mys.fail()) {failBadFormat(str);}
     return ret;
   }
```

```
   int DateTime::eatChars(std::istream & is, int nchars) const {
     // consume nchars characters from the stream and interpret as an integer
+    if (nchars < 0) ABORT("Cannot read a negative number of characters.");
+    std::string str((size_t) nchars, '\0');
+    is.get(&str[0], nchars+1);  // nchars+1 because istream.get reads (count-1) chars.

+    int ret = 0;
+    try {
+      ret = boost::lexical_cast<int>(str);
+    }
+    catch (boost::bad_lexical_cast&) {
+      failBadFormat(str);
+    }
     return ret;
   }
```

Console-based usage

- amplxe-cl –collect hotspots –result-dir out –quiet -- your_app_here.x arg1 arg2 …
- If –quiet is not specified, a summary report is printed to the console.
- The results directory is around 5-10 MB per unit test. Can be transferred between computers.
- Not restricted to profiling an application. Can also use a script or mpiexec.