Performance monitors and profilers

Scott Snyder

Brookhaven National Laboratory, Upton, NY, USA

Mar 21, 2019
HOW2019, JLab

Introduction

Questions like

- How can I make my code run faster?
- How can I reduce the amount of memory my code needs?
- Is my code making good use of capabilities of the hardware?

are usually best addressed using tools that can measure the performance of the code.

Measuring resources required by individual fragments of code is called *profiling*.

Types of profiling

Statistical sampling

Interrupt program periodically with a timer. Record/histogram the program counter at each interrupt.
[vtune, oprofile, etc.]

Performance counters

Modern CPUs maintain a variety of performance counters: instructions executed, cache misses, etc. Can collect detailed information with low overhead.
Often require root privileges to access.
[vtune, oprofile, etc.]

Simulation/emulation

Simulate program execution on a virtual machine, and collect performance information from the model machine.
Slow. [valgrind (sort of...)]

Types of profiling

Manual instrumentation

Explicitly add calls to measure resources used by a particular piece of code.
Frameworks generally have support for, eg, measuring time and memory used on a per-algorithm basis.

Automatic instrumentation

Code is transformed automatically to include performance analysis.
Can be done by the compiler, or by a separate tool.
Can also intercept and instrument library calls to monitor, eg, memory usage.
[gprof, valgrind, etc.]

Event tracing

Allows program to log interesting events as a function of time for later analysis. May also be useful for debugging.
[vtune, perf, tau]

Further considerations

Flat vs. call-stack profiling

Some profilers collect only the program counter. But if a program is spending a lot of time in, say, `memcpy`, then you want to know which functions are making most of the calls. Profilers that collect call stack information can help with this.
Different tools may collect the stack to different depths, and may also differ in handling cycles int the call graph.

Overhead

Collecting data will add to runtime and memory requirements. Can vary from ~ negligible to over an order of magnitude.

Bias

The act of collecting performance data can bias the performance results. Some tools try to correct for this, more or less successfully. Somewhat correlated with overhead.

Further considerations

Statistics

Tools relying on statistical sampling need the program to run for long enough to accumulate sufficient statistics. Tools relying on instrumentation may be fine with shorter runs.

Control

It is often useful to be able to restrict data collection to some phases of the program — for example, collect data only for event processing but not for initialization for finalization.
Can be either interactive or programmatic. No real standard on how to do this.

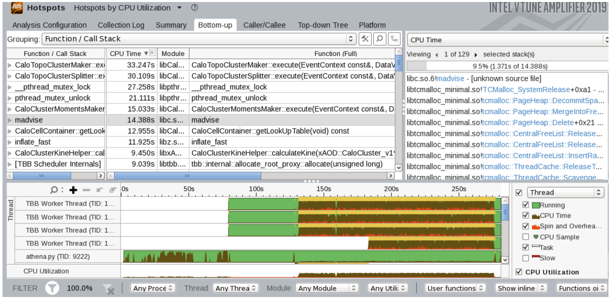
User interface

What tools are available to inspect and analyze the results?
Can vary from a simple textual dump to an elaborate GUI.

Intel VTune

- Comprehensive; low-overhead.
- (Usually) works well for very large applications.
- Timeline analysis.
- Support for finding issues due to locking, etc.

- Proprietary.
- Need root access to go much beyond basic statistical sampling.
- Sometimes crashes for complicated applications.



Function / Call Stack	CPU Time ▼	Module	Function (Full)
CaloTopoClusterMaker::ex	33.247s	libCal...	CaloTopoClusterMaker::execute(EventContext const&, DataV
CaloTopoClusterSplitter::ex	30.109s	libCal...	CaloTopoClusterSplitter::execute(EventContext const&, Data
__pthread_mutex_lock	27.258s	libpthr...	__pthread_mutex_lock
pthread_mutex_unlock	21.111s	libpthr...	pthread_mutex_unlock
CaloClusterMomentsMaker	15.033s	libCal...	CaloClusterMomentsMaker::execute(EventContext const&, D
madvise	14.388s	libc.s...	madvise
CaloCellContainer::getLook	12.955s	libCal...	CaloCellContainer::getLookUpTable(void) const
inflate_fast	11.925s	libz.s...	inflate_fast
CaloClusterKineHelper::cal	9.450s	libxA...	CaloClusterKineHelper::calculateKine(xAOD::CaloCluster_v1
[TBB Scheduler Internals]	9.039s	libtbb....	tbb::internal::allocate_root_proxy::allocate(unsigned long)

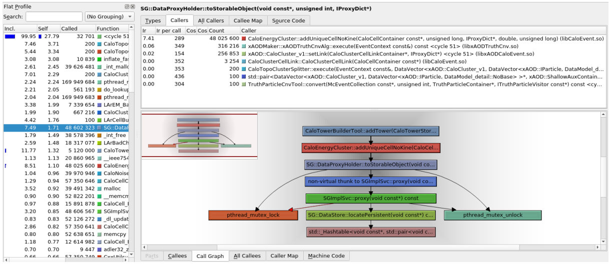
```
Viewing 1 of 129 selected stack(s)
9.5% (1.371s of 14.388s)
libc.so.6!madvise - [unknown source file]
libtcmalloc_minimal.so!TCMalloc_SystemRelease+0xa1 - ...
libtcmalloc_minimal.so!tcmalloc::PageHeap::DecommitSpa...
libtcmalloc_minimal.so!tcmalloc::PageHeap::MergeIntoFre...
libtcmalloc_minimal.so!tcmalloc::PageHeap::Delete+0x21 ...
libtcmalloc_minimal.so!tcmalloc::CentralFreeList::Release...
libtcmalloc_minimal.so!tcmalloc::CentralFreeList::Release...
libtcmalloc_minimal.so!tcmalloc::CentralFreeList::InsertRa...
libtcmalloc_minimal.so!tcmalloc::ThreadCache::ReleaseT...
libtcmalloc_minimal.so!tcmalloc::ThreadCache::Scavenge...
```

Valgrind

- Open source; available in most linux distributions.
- Very robust.
- No root privs needed.
- Nice GUI for examining results (kcachegrind).
- Also a variety of runtime checking.

Dynamically transforms machine code to add instrumentation.

- Large overhead (10–50×).
- Results sometimes significantly biased.



Flat Profile

Search: (No Grouping)

Incl.	Self	Called	Function
99.95	27.79	32 701	<cycle 51
7.46	3.71	200	CaloTopo
5.44	3.34	200	CaloTopo
3.08	3.08	10 839	inflate_fa
2.61	2.45	39 626 481	_int_mallo
7.01	2.29	200	CaloClust
2.24	2.24	169 949 684	pthread_r
2.21	2.05	561 193	do_lookup
2.04	2.04	169 949 683	pthread_r
3.38	1.99	7 339 654	LArEM_Ba
1.99	1.90	667 216	CaloClust
4.42	1.76	100	LArCellBu
7.49	1.71	48 602 323	SG::DataP
1.79	1.49	38 578 396	_int_free
2.59	1.48	18 317 077	LArBadCh
11.77	1.32	5 120 000	CaloTowe
1.13	1.13	20 860 965	__ieee754
8.51	1.10	48 025 600	CaloEnerg
1.04	0.96	39 970 946	CaloNoise
1.29	0.94	57 350 646	CaloCellC
3.52	0.92	39 491 342	malloc
0.90	0.90	52 822 201	__memcm
0.97	0.88	15 891 878	CaloCell_I
3.20	0.85	48 606 567	SGImplSv
0.83	0.83	52 126 272	_dl_updat
2.86	0.82	57 350 641	CaloCellC
0.80	0.80	52 638 651	memcpy
1.18	0.77	12 614 982	CaloCell_I
0.70	0.70	9 447	adler32_z
0.66	0.66	57 350 740	CxxUtils:

SG::DataProxyHolder::toStorableObject(void const*, unsigned int, IProxyDict*)

Types | Callers | All Callers | Callee Map | Source Code

Ir	Ir per call	Cos	Cos	Count	Caller
7.41	289			48 025 600	CaloEnergyCluster::addUniqueCellNoKine(CaloCellContainer const*, unsigned long, IProxyDict*, double, unsigned long) (libCaloEvent.so)
0.06	349			316 216	xAODMaker::xAODTruthCnvAlg::execute(EventContext const&) const <cycle 51> (libxAODTruthCnv.so)
0.02	154			256 853	xAOD::CaloCluster_v1::setLink(CaloClusterCellLinkContainer*, IProxyDict*) <cycle 51> (libxAODCaloEvent.so)
0.00	352			3 254	CaloClusterCellLink::CaloClusterCellLink(CaloCellContainer const*) (libCaloEvent.so)
0.00	353			200	CaloTopoClusterSplitter::execute(EventContext const&, DataVector<xAOD::CaloCluster_v1, DataVector<xAOD::IParticle, DataModel_d...
0.00	436			100	std::pair<DataVector<xAOD::CaloCluster_v1, DataVector<xAOD::IParticle, DataModel_detail::NoBase> >*, xAOD::ShallowAuxContain...
0.00	304			100	TruthParticleCnvTool::convert(McEventCollection const*, unsigned int, TruthParticleContainer*, ITruthParticleVisitor const*) const <cy...

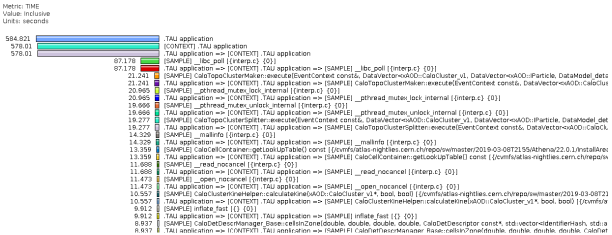
CaloTowerBuilderTool::addTower(CaloTowerStor...
CaloEnergyCluster::addUniqueCellNoKine(CaloCel...
SG::DataProxyHolder::toStorableObject(void co...
non-virtual thunk to SGImplSvc::proxy(void co...
SGImplSvc::proxy(void const*) const
pthread_mutex_lock
SG::DataStore::locatePersistent(void const*) c...
pthread_mutex_unlock
std::_Hashtable<void const*, std::pair<void c...
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code

TAU (Tuning and analysis utilities)

- Very comprehensive: statistical counters, performance counters, optional instrumentation, GPUs, MPI, etc. Muiltilanguage!
- Open source.
- No root privs needed.
- Developers eager for collaboration.

- Not so easy to figure out how to use.
- Some small (fixable) bugs on large applications.
- Analysis tools confusing. Some parts, like call graph display, are simply useless for large applications.

See backup for further details.



```
Metric: TIME
Value: Inclusive
Units: seconds

584.821  .TAU application
 578.01  [CONTEXT] .TAU application
 578.01  .TAU application => [CONTEXT] .TAU application
         87.178  [SAMPLE] __libc_poll [{interp.c} {0}]
         87.178  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __libc_poll [{interp.c} {0}]
          21.241  [SAMPLE] CaloTopoClusterMaker::execute(EventContext const&, DataVector<xAOD::CaloCluster_v1, DataVector<xAOD::IParticle, DataModel_deta
          21.241  .TAU application => [CONTEXT] .TAU application => [SAMPLE] CaloTopoClusterMaker::execute(EventContext const&, DataVector<xAOD::CaloClus
          20.965  [SAMPLE] __pthread_mutex_lock_internal [{interp.c} {0}]
          20.965  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __pthread_mutex_lock_internal [{interp.c} {0}]
          19.666  [SAMPLE] __pthread_mutex_unlock_internal [{interp.c} {0}]
          19.666  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __pthread_mutex_unlock_internal [{interp.c} {0}]
          19.277  [SAMPLE] CaloTopoClusterSplitter::execute(EventContext const&, DataVector<xAOD::CaloCluster_v1, DataVector<xAOD::IParticle, DataModel_det
          19.277  .TAU application => [CONTEXT] .TAU application => [SAMPLE] CaloTopoClusterSplitter::execute(EventContext const&, DataVector<xAOD::CaloClu
          14.329  [SAMPLE] __mallinfo [{interp.c} {0}]
          14.329  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __mallinfo [{interp.c} {0}]
          13.359  [SAMPLE] CaloCellContainer::getLookUpTable() const [{/cvmfs/atlas-nightlies.cern.ch/repo/sw/master/2019-03-08T2155/Athena/22.0.1/InstallArea
          13.359  .TAU application => [CONTEXT] .TAU application => [SAMPLE] CaloCellContainer::getLookUpTable() const [{/cvmfs/atlas-nightlies.cern.ch/repo/sw
          11.688  [SAMPLE] __read_nocancel [{interp.c} {0}]
          11.688  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __read_nocancel [{interp.c} {0}]
          11.473  [SAMPLE] __open_nocancel [{interp.c} {0}]
          11.473  .TAU application => [CONTEXT] .TAU application => [SAMPLE] __open_nocancel [{interp.c} {0}]
          10.557  [SAMPLE] CaloClusterKineHelper::calculateKine(xAOD::CaloCluster_v1*, bool, bool) [{/cvmfs/atlas-nightlies.cern.ch/repo/sw/master/2019-03-08T21
          10.557  .TAU application => [CONTEXT] .TAU application => [SAMPLE] CaloClusterKineHelper::calculateKine(xAOD::CaloCluster_v1*, bool, bool) [{/cvmfs/a
           9.912  [SAMPLE] inflate_fast [{} {0}]
           9.912  .TAU application => [CONTEXT] .TAU application => [SAMPLE] inflate_fast [{} {0}]
           8.937  [SAMPLE] CaloDetDescrManager_Base::cellsInZone(double, double, double, double, CaloDetDescriptor const*, std::vector<IdentifierHash, std::a
           8.937  .TAU application => [CONTEXT] .TAU application => [SAMPLE] CaloDetDescrManager_Base::cellsInZone(double, double, double, double, CaloDet
```

Others

- gprof: Requires recompilation to add instrumentation. Generally gives accurate information even for short runs, but doesn't work with shared libraries.
- `LD_PROFILE`: Dynamic loader can profile a single shared library.
- oprofile: Low-overhead, system-wide profiler. Requires root.
- gooda: Developed with HEP applications in mind, but seems to be dead?
- igprof: Was useful in the past, but didn't succeed in making it work with Athena on the last attempt.
- google-perftools: Includes statistical profiling. Was previously integrated with Athena scripts. But doesn't seem to work correctly with Athena recently (most call arcs are missing).
- Other proprietary tools: Vampir, Apple, AMD, PGI, nvidia, etc.

Summary

- Tools that I mostly use currently are valgrind and vtune.
- TAU looks to have promise, but needs a bit of work before being really useful.
    - Something to convert TAU profiling data to kcachegrind format might be a real help in getting people started!
- Other commercial tools may be useful on specific platforms (but I don't have personal experience with them).

TAU notes

- Available from `https://www.cs.uoregon.edu/research/tau/home.php`
- Probably need to also download binutils, etc., as mentioned in the build instructions.
- Important to build TAU with the same compiler as used for the application being profiled.
- Can collect data using '`tau_exec -ebs COMMAND`'.
  - `-ebs` means to do statistical sampling. If you leave it off, everything will appear to work ok, but there will be no data in the resulting profile.
- If `fork()` takes too long to execute, TAU can enter an infinite loop. Need to disable/block SIGPROF around the fork().