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 in the call graph.

Overhead

Collecting data will add to runtime and memory requirements. Can vary from \( \sim \) 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.
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.
TAU (Tuning and analysis utilities)

- Very comprehensive: statistical counters, performance counters, optional instrumentation, GPUs, MPI, etc. Multilanguage!
- 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.

Scott Snyder (BNL)
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.
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().