Measurement: Overview

Vojtěch Horký   Peter Libič   Petr Tůma

2010 – 2023

Contents

Events and States

Events

Typically, performance metrics talk about events. For example:

- Operation (or method or function or transaction) start (finish).
- Network packet transmission (reception).
- Memory reference.
- Disk access.
- ...

States

Alternatively, performance metrics may relate to system state. Events mark system state changes.

Metrics

Metrics are related to events and states in several typical ways:

- *Event-Count or Event-Time Metrics* A metric counts or times event occurrences (dynamic instruction count, I/O throughput, time tick ...).
- *Secondary Event Metrics* A metric tracks state attributes on event occurrences (packet sizes, exception locations, calling contexts ...).
- *Event Profiles* A metric creates overall system profile (execution time distribution, utilization ...).

Measurement Approaches

- *Event Driven Measurement* Getting control to collect metric information on every event (for example mechanism to increment (hardware) counter on cache miss).
- *Trace Collection* Recording a trace of events together with useful parts of system state (for example collecting stack dump for method invocation trace).
- *Sampling* Recording system state at certain intervals to estimate the metric of interest (for example collecting execution profile).
- *Indirect* Deriving the value of a metric that is difficult (or impossible) to measure from other metric(s) (for example collecting object lifetimes).

Two Big Problems with Measurement

Overhead

Measurement can incur (possibly very big) overhead:

- time executing measurement code
- consumed storage space or network bandwidth
- consumed program memory or system resources
- ...

Perturbation

Measurement can change observed system behavior:

- including measurement overhead in measurements
- changing behavior during execution
    - synchronization artefacts
    - changes in optimization
    - ...

Example: Event-Driven Interval Measurement

We often ask what happened during an operation:

- How long did it take ?
- How many cache misses happened while the operation executed ?

In general, simple code does the job:

```
before = read_value ();
// run the operation of interest
after = read_value ();
results.append (after - before);
```

Let us look at potential for overhead and perturbation.

Interval Measurement Memory Overhead

The results storage consumes memory which means:

- Application has less available memory. Could cause swapping or behavior change (less memory for buffers ...)
- Application has less cache available to it. Changes amount of (capacity) cache misses.
- Application has different memory layout. Changes amount of (conflict) cache misses.
- Higher (or different) load for memory allocator or garbage collector.

In some cases reducing memory overhead by these methods can help:

- Compress data. Even simple methods such as using fewer bits.
- Process data immediately. Online computation of metrics such as average possible.
- Use cache-friendly patterns to store the data.

Interval Measurement Time Overhead

```
before = read_value ();
// run the operation of interest
after = read_value ();
results.append (after - before);
```

If measured value is time, part of code above is included in result:

- Reading and recording the *before* value.
- Call and return to (from) the operation.
- Reading the *after* value.

Of course, other metrics (like cache misses) may have similar overhead. This overhead might be measured and compensated, but it is very difficult.

Interval Measurement Perturbations

More complex perturbation effects possible:

- Time to store results may affect scheduling and alter minute timing of later measurements.
- With measurement included, optimizations might be reduced (register allocation, pipelining, branch prediction …).
- More function calls also affect execution (safe points, stack …).

Interval Measurement Perturbations

Important questions:

- Can this be avoided ?
- How much does this matter ?

Initial measurement guidelines:

- Keep measurement code as simple as possible. Especially avoid conditional branching. Same path for warmup and measurement.
- Measure only operations taking long time (large counter values).
- The measured operation should take at least 100-1000 times longer than the measurement overhead.

> To illustrate the issues of overhead and perturbation, consider the task of measuring object lifetimes in Java. Determining whether an object is live is the task of the garbage collector. A garbage collection cycle is expensive to execute and therefore infrequent. Recording object lifetimes during garbage collection cycles necessarily rounds each lifetime to the nearest garbage collection cycle, losing accuracy. It is possible to execute the garbage collection cycle more frequently, increasing the accuracy but also the overhead, up to potentially prohibitive levels.
> A more sophisticated approach, described by Hertz et al.: Generating Object Lifetime Traces … doi:10.1145/1133651.1133654, requires instrumenting reference updates to reduce the overhead – it is reported to be up to 800 times faster than approaches that execute garbage collection cycles frequently, but still 70 to 300 times slower than normal execution. A more recent version of the same approach, based on efficient recording of object allocation events, has been evaluated by Lengauer et al.: Accurate and Efficient Object Tracing … doi:10.1145/2668930.2688037 and shown to reduce the average overhead to around 5 % at the cost of extensive virtual machine modifications.
> When instrumenting reference updates, the measurement may further introduce perturbation by passing around object references. The virtual machine compiler may track the object references to perform escape analysis, which can identify objects restricted to local (method) scope, and allocate such objects on the stack. The instrumentation may influence the escape analysis and thus increase heap allocation. For details see Zheng et al.: Accurate Profiling … doi:https://doi.org/10.1145/2814270.2814281.