semile

profile what you care, monitor how it goes

View the Project on GitHub r-kan/semile

Download **ZIP File**	Download **TAR Ball**	View On **GitHub**

semile :)

Why semile?
We semile bcoz it helps overcome the flaw/bottleneck of our programs

What is *semile*?

A profiling framework that provides the ability to monitor programs, in general of any programming language, by the following two pieces of information:
1. consumed time per execution
2. 'footprint' message per execution

Difference with other profiling tools?

- ***Profile 'semantically'*** Each call to the same function plays its individual role within profiling. Normal 'syntactic' profilers are good in other aspects but fail to achieve this.
- ***Lightweight*** The profiled program gives little run-time overhead. The viewer is compact that targets to provide only necessary information without fancy visual effect. It gives profile result in widespread PNG and XML format.
- ***Message-embedded profile*** Custom information can be left within profile elements. It then also provides the ability to help reveal internal state/decision inside the program.

P.S. The user-provided semantic specifications (via the profile library) is necessary for semantic profile

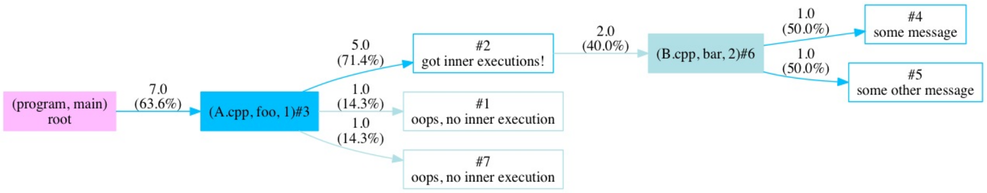
System Requirement

python3 (viewer)
g++ (cpp profile library)

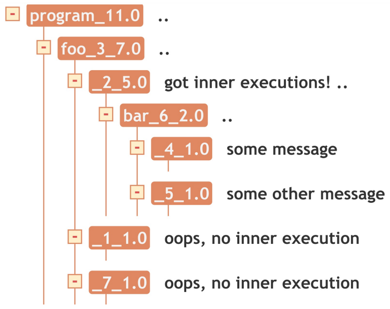
Dependent Library

dot (graphviz)

Viewer Demonstration



(PNG format)



```
program_11.0 ..
  foo_3_7.0 ..
    _2_5.0 got inner executions! ..
      bar_6_2.0 ..
        _4_1.0 some message
        _5_1.0 some other message
    _1_1.0 oops, no inner execution
    _7_1.0 oops, no inner execution
```

(XML format, browse by codebeautify.org/xmlviewer))

Profile Library

`semile` does not aim to profile all program execution, it only profile the execution specified via the profile library. Currently, cpp profile library is provided.

Tutorial: profile a cpp program

1. Choose the statements to be profiled
2. Let the statements be in some sense derived from `ExecutionMonitor`
   - Model the statements as a function, and place it inside an `ExecutionMonitor` descendant.
   - In particular, if the statements matches life cycle of a class object, then subclass `ExecutionMonitor` does the job.

Optionally, call `ExecutionMonitor::addMessage()` during life cycle of the `ExecutionMonitor` descendant, to leave custom footprint message.

Code example

```
void quicksort(vector<int>& x, int start_pos, int end_pos);
```

Suppose *quicksort* is the profiling target,

```
class QuicksortMonitor: public ExecutionMonitor
{
  QuicksortMonitor()
   :ExecutionMonitor("quicksort", __FILE__, __LINE__)  {}
  void operator()(vector<int>& x, int start_pos, int end_pos)
  {
    return quicksort_impl(x, start_pos, end_pos);
  }
};
```

Class *QuicksortMonitor*, derived from `ExecutionMonitor`, is created. There is a function operator inside *QuicksortMonitor*, with its interface and implementation copy from *quicksort*. Note that *quicksort* is renamed to *quicksort_impl*.

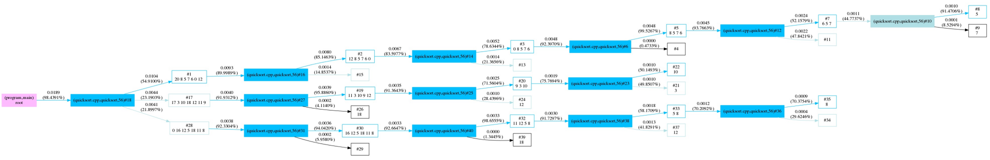
```
void quicksort(vector<int>& x, int start_pos, int end_pos)
{
  QuicksortMonitor()(x, start_pos, end_pos);
}
```

The profiling enabled *quicksort* now instantiate a *QuicksortMonitor* instance, and invokes its function operator.

```
void QuicksortMonitor::addMsg(
  const vector<int>& x, int start_pos, int end_pos)
{
  stringstream stream;
  for (int i = start_pos; i <= end_pos; ++i) {
    stream << x[i] << " ";
  }
  addMessage(stream.str());
}
```

In addition, we can log profile message within *QuicksortMonitor* at any time.

One possible viewer generated PNG is as follows:
(run quicksort 3 times with random inputs)



Contact

Please contact *Rodney Kan* by its_right@msn.com for any question/request/bug without hesitation.

This project is maintained by r-kan