Enabling HPC software productivity with the TAU performance system

Jean-Baptiste BESNARD
<jbbesnard@paratools.fr>

PDC Summer School August 2023, KTH, Stockholm, Sweden.

Instrumentation

Various means of capturing program’s state

Direct Observation

Direct Performance Observation

Execution *actions* exposed as *events*

- In general, actions reflect some execution state
  - presence at a code location or change in data
  - occurrence in parallelism context (thread of execution)
- Events encode actions for observation

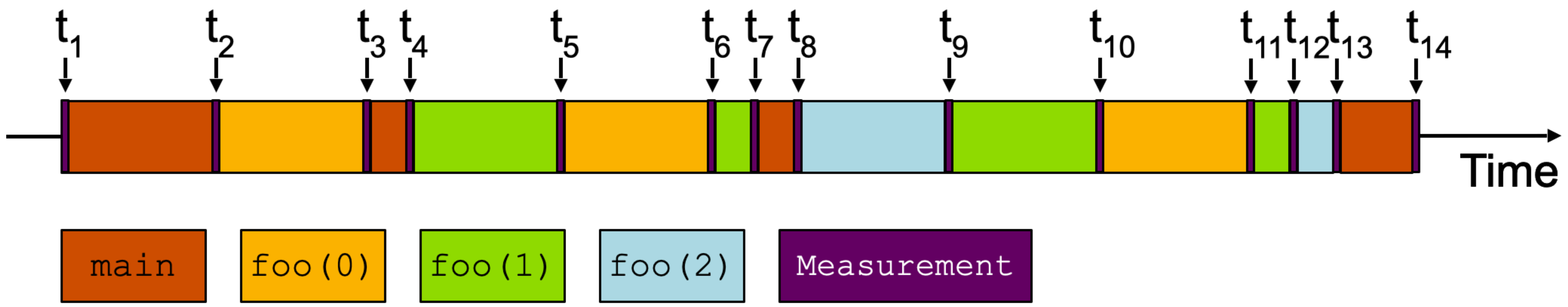
Observation is *direct*

- Direct instrumentation of program code (probes)
- Instrumentation invokes performance measurement
- Event measurement = performance data + context

Performance experiment

- Actual events + performance measurements

Instrumentation



Measurement code is inserted such that every event of interest is captured directly

- Can be done in various ways

Advantage:

- Much more detailed information

Disadvantage:

- Processing of source-code / executable necessary
- Large relative overheads for small functions

```
int main()
{
   int i;
    TAU_START("main");
   for  (i=0; i < 3; i++)
     foo(i);
   TAU_STOP("main");
   return 0;
}

void foo(int i)
{
    TAU_START("foo");

   if  (i > 0)
     foo(i - 1);
   TAU_STOP("foo");
}
```

Three Instrumentation Techniques for Wrapping External Libraries

Pre-processor based substitution by re-defining a call (e.g., read)

Preloading a library at runtime

Linker based substitution

Preprocessor based substitution

Pre-processor based substitution by re-defining a call

- Compiler replaces read() with tau_read() in the body of the source code

Advantages:

- Simple to instrument
  - Preprocessor based replacement
  - A header file redefines the calls
  - No special linker or runtime flags required

Disadvantages

- Only works for C & C++ for replacing calls in the body of the code.
- Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., libhdf5.a)

Preloading a wrapper library

Preloading a library at runtime

- Tool defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read
- *tau_exec* tool uses this mechanism to intercept library calls

Advantages

- No need to re-compile or re-link the application source code
- Drop-in replacement library implemented using LD_PRELOAD environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris…

Disadvantages

- Only works with dynamic executables. Default compilation mode under Cray XE6 and IBM BG/P is to use static executables
- Not all operating systems support preloading of dynamic shared objects (DSOs)

Linker based substitution

Linker based substitution

- Wrapper library defines __wrap_read which calls __real_read and linker is passed -Wl,-wrap, read

Advantages

- Tool can intercept all references to a given call
- Works with static as well as dynamic executables
- No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library

Disadvantages

- Wrapping an entire library can lengthen the linker command line with multiple –Wl,-wrap,<func> arguments. It is better to store these arguments in a file and pass the file to the linker
- Approach does not work with un-instrumented binaries

Indirect Performance Observation

Program code instrumentation is not used

Performance is observed indirectly

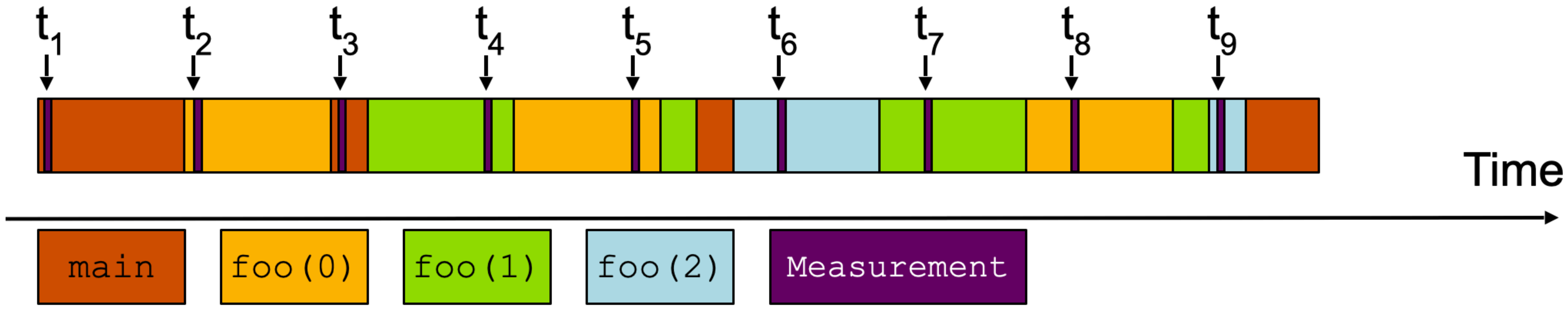
- Execution is interrupted
  - can be triggered by different events
- Execution state is queried (sampled)
  - different performance data measured
- *Event-based sampling* (EBS)

Performance attribution is inferred

- Determined by execution context (state)
- Observation resolution determined by interrupt period
- Performance data associated with context for period

Indirect Observation

Sampling



- Addresses are mapped to routines using symbol table information

Statistical inference of program behavior

- Not very detailed information on highly volatile metrics
- Requires long-running applications

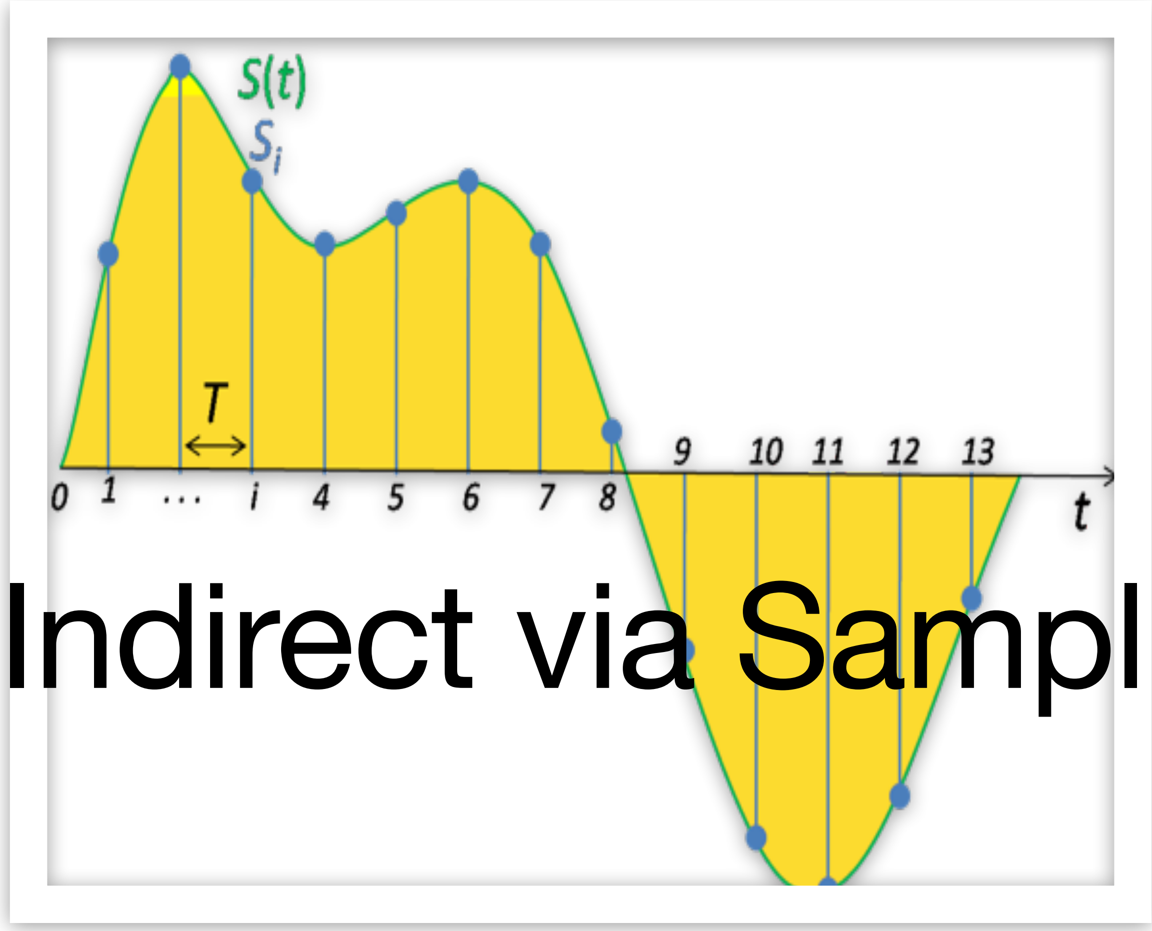
Works with unmodified executables

```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

Performance Data Measurement

```
Call
START('potential')
// code
Call
STOP('potential')
```

- Exact measurement
- Fine-grain control
- Calls inserted into code

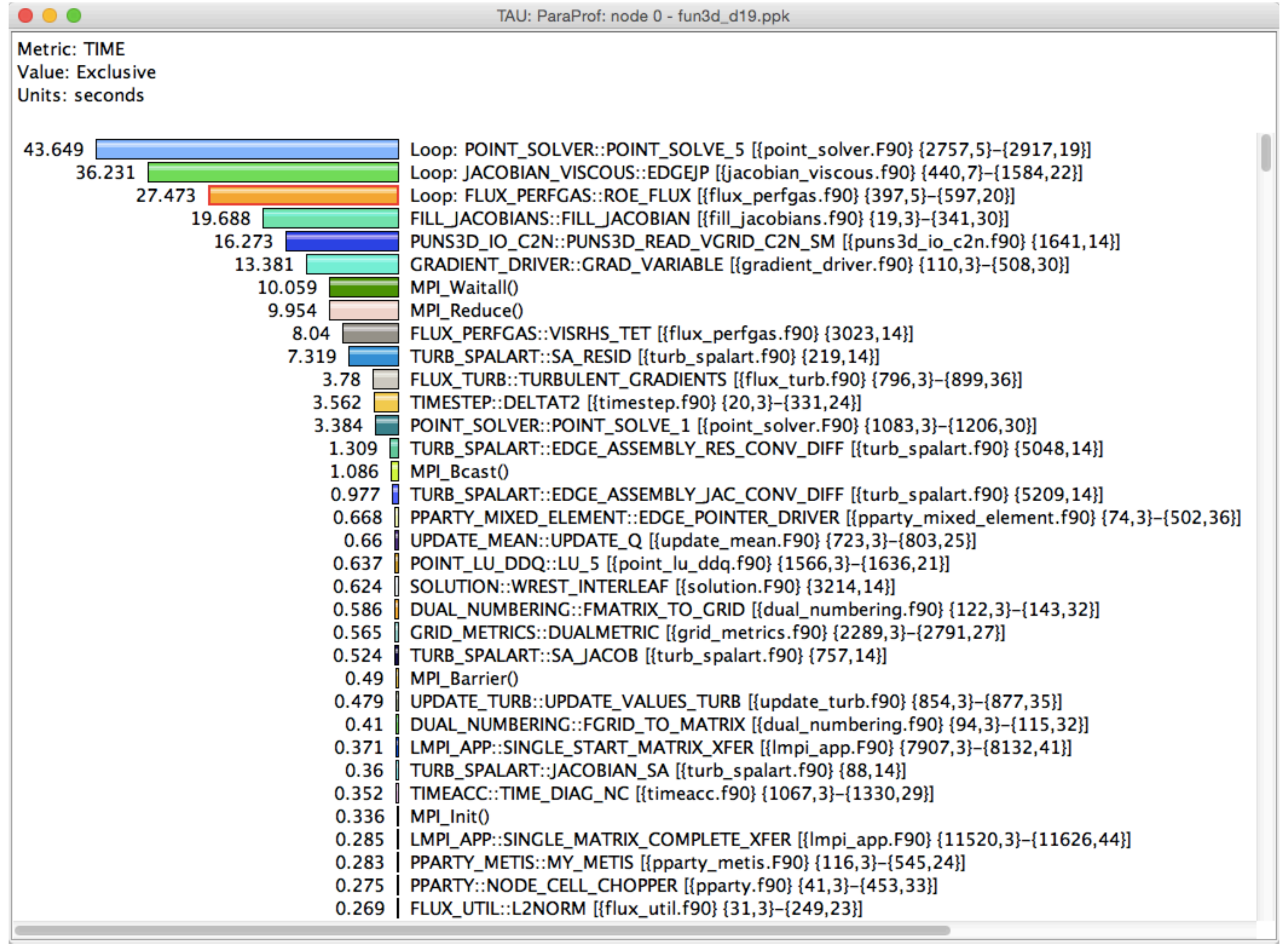


Indirect via Sampling

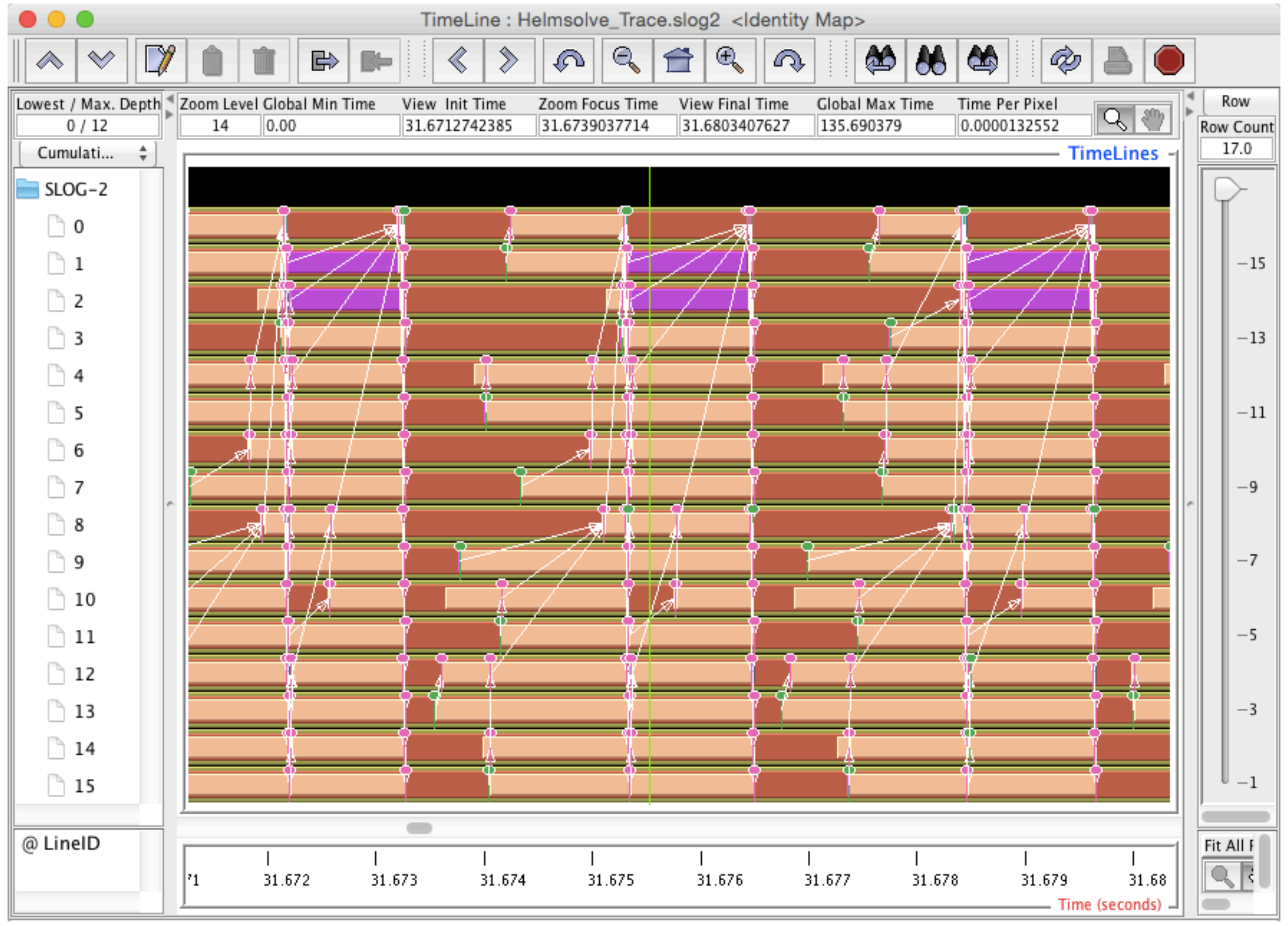
- No code modification
- Minimal effort
- Relies on debug symbols (**-g**)

Measurement Verbosity

Profiling and Tracing



Profiling

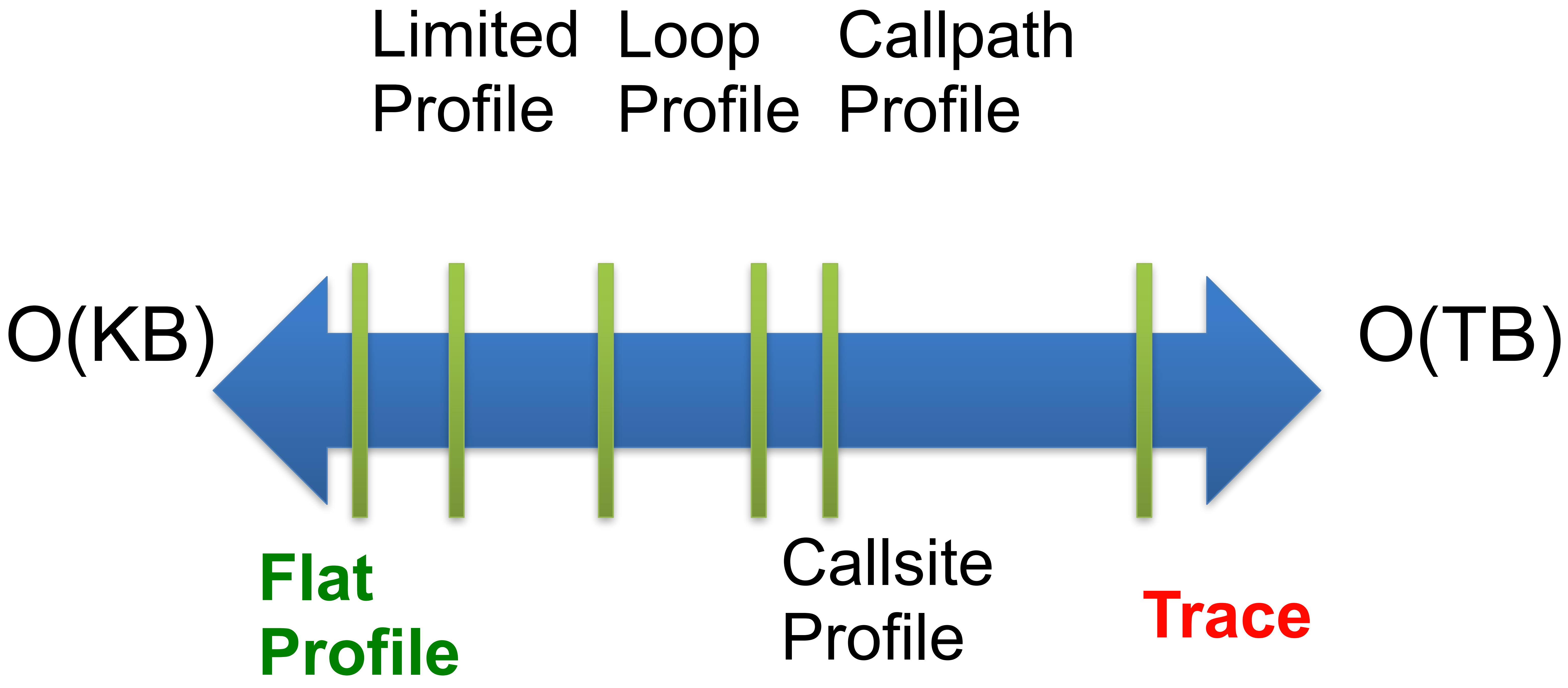


Tracing

Profiling shows you **how much** (total) time was spent in each routine

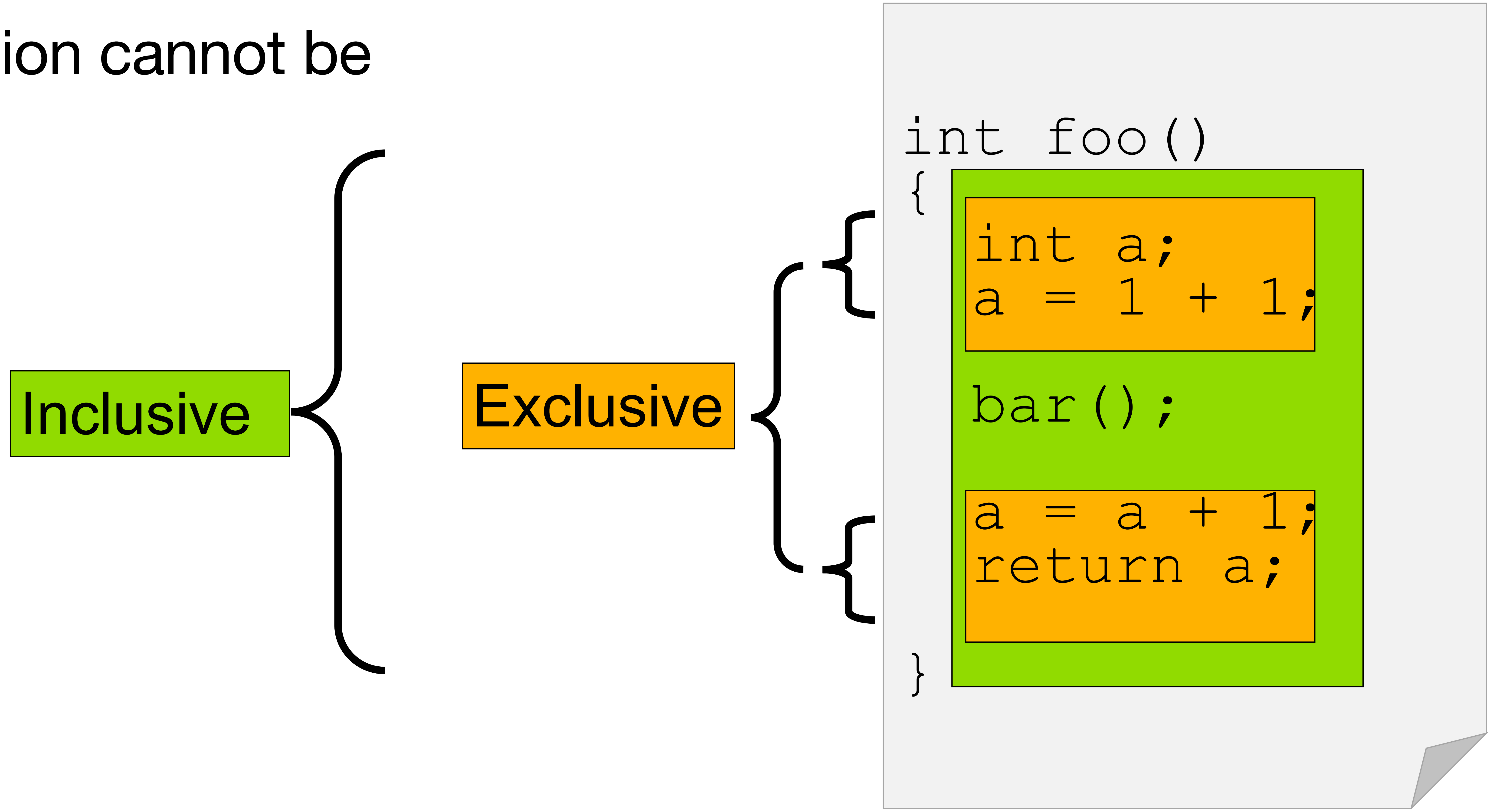
Tracing shows you **when** the events take place on a timeline

How much data do you want?

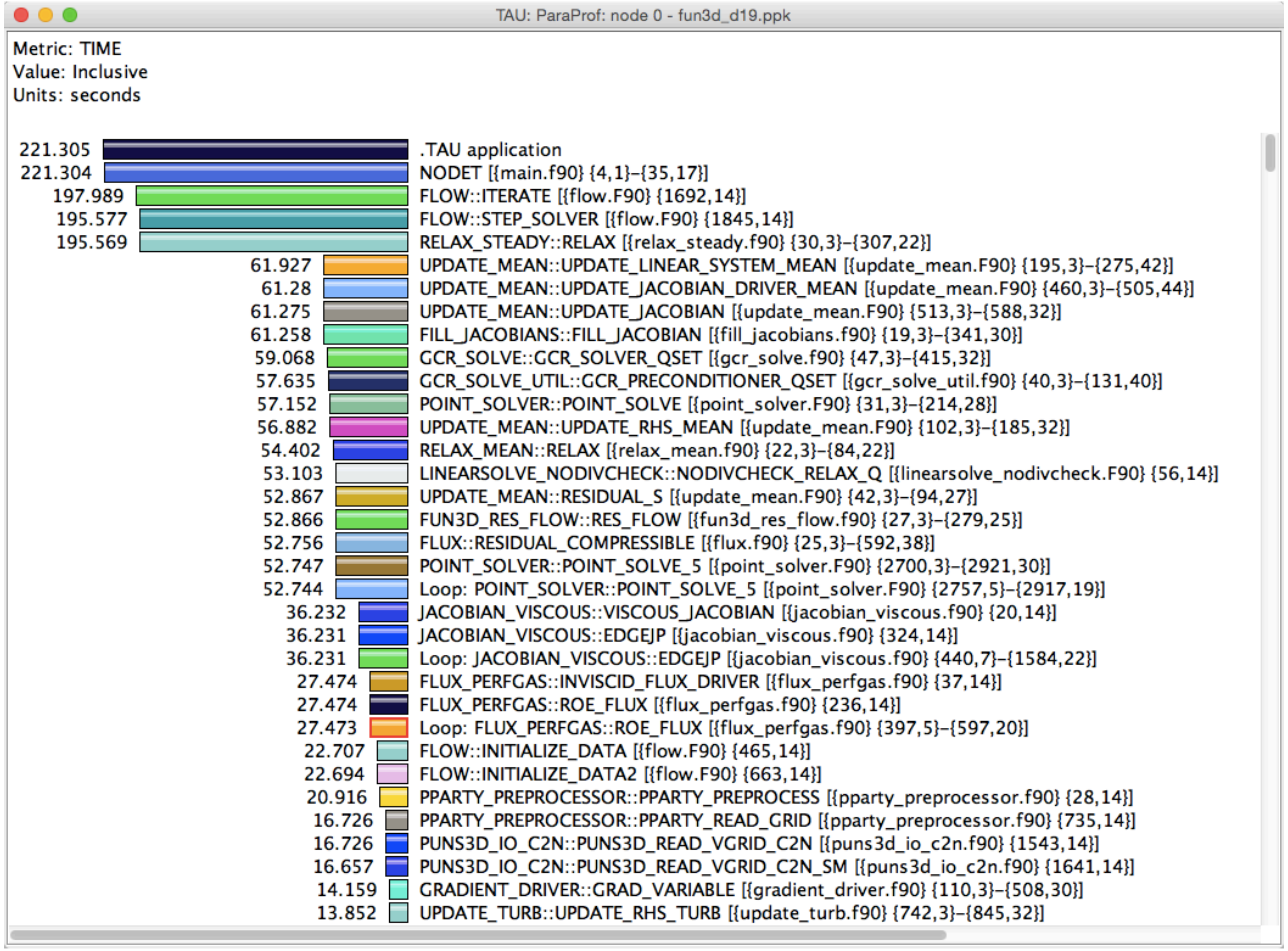


Inclusive vs. Exclusive values

- **Inclusive**: Information of all sub-elements aggregated into single value
- **Exclusive**: Information cannot be subdivided further



Inclusive Measurements



Exclusive Time