COMP 161 - Lecture Notes - 21 - Profiling part 2

Spring 2014

In these notes we look at profiling techniques that require additional code to be added to a program. These techniques allow for the collection of a more focused, specific data set than we can get from using valgrind on unmodified code. They also expose us to techniques for getting very fine grained execution time data. These profiling techniques use the callgrind macros defined in *callgrind.h* library and the new C++11 *chrono* library.

Invasive Profiling

Our previous efforts at profiling code did not require any changes to the code that we wished to profile. All tAt most, we'd write some contrived *main* procedures to make specific calls to the target code. Profiling without modifying existing library code allows us to profile a program in context. Often this lets you get a good idea of what's going on under the hood. On the other hand, it produces data about a lot of code we're not concerned with and can prevent us from getting the kind of detailed data we're looking for.

Sometimes we're not worried about code within a specific application, but instead we're focused on detailed experimentation. For example, to get a good picture of the behavior of a specific vector procedure we're likely to want to run the procedure on a large number of inputs and gather individualized data about each execution.

Getting Execution time with chrono

The general pattern for timing code is to check the time prior to code execution and then again after code execution. You then subtract the start time from the end time to get the elapsed duration. You can then add time to other times and average the result as needed. The C++11 library *chrono* provides several classes that can be used to time the execution of code. There is a complete example for getting the execution time of a line of code in the example for the *now* function, which gets the current time[1]

To get the current time we'll use a *high_resolution_clock::time_point* object. As the name indicates, these objects measure time at a high resolution[2]. We can store the difference of two time points as a *duration<double>* which, by default, tells us the number of seconds as a double. We can access the number of seconds with the duration class method, *count*. So, if we wanted to time *foo* and print the time taken to execute foo, then we might do the following:

[1] `http://www.cplusplus.com/reference/chrono/high_resolution_clock/now/`

[2] nano seconds

```
// make time_points for start and end time
```

```
high_resolution_clock::time_point start;
high_resolution_clock::time_point end;
// make duration<double> for elapsed time
duration<double> elapsed;

//get start time
start = high_resolution_clock::now();
//run foo
foo();
//get end time
end = high_resolution_clock::now();

//compute elapsed time as duration
elapsed = duration_cast< duration<double> >(end-start);

//print elapsed time (in seconds)
cout << "Foo took " << elpased.count() << " secs.\n";
```

This code is simply a modification of the example found in the documentation for *high_resolution_time::now*. What it illustrates is that the time oriented classes provide all the functionality we need for getting execution time and that we can even do things like subtract time points using *operator-*. Operators also work with *duration<double>*, so you can add, subtract, multiple and divide durations as needed[3].

[3] See `http://www.cplusplus.com/reference/chrono/duration/operators/`

Using callgrind.h to get data programmatically

Valgrind provides a C/C++ library called *callgrind.h* that allows you to specify, within the code, when callgrind should stop and start data collection as well as when callgrind should produce a data file. This means that we can get callgrind to collect data for specific lines of code and dump a data file only for that line of code. First off, you must include the library *valgrind/callgrind.h* in order to control callgrind from within your program. We'll focus on telling callgrind to turn all instrumentation off when not needed. This can lead to faster profiling. Alternatively, we could leave instrumentation on and stop and start data collection[4].

[4] see `http://stackoverflow.com/q/13688185/1042494` for some discussion.

Let's see what it might look like when we want to gather individual callgrind data about the execution of five separate executions of *foo* within a program's *main* procedure.

```
#include <valgrind/callgrind.h>

int main(int argc, char* argv[]){
```

```
 // code

 for(int i(0); i<5; i++){
   CALLGRIND_START_INSTRUMENTATION;
   foo();
   CALLGRIND_STOP_INSTRUMENTATION;
   CALLGRIND_DUMP_STATS;
 }

 // more code

}
```

When we run this program with callgrind, then callgrind effectively turns itself on prior to the execution of foo and shuts itself off after foo. The dump stats statement, forces valgrind to write a data file of the data collected so far and then reset all counters. The result would be 6 different callgrind data files. They'd all have the same prefix with the same process id, callgrind.data.*pid*, but the five data dumps corresponding to the five dump stats statements would have the numbers 1 to 5 appened to it. So callgrind.data.*pid*.1 would be the file corresponding to first forced stats dump. The sixth file is all the other stuff collected, which in our case would be nothing. To use the callgrind control statements we must start callgrind without instrumentation. This is done with the callgrind option *–instr-atstart=no*. For further details, consult the callgrind documentation.

There is also a way to specify the reason for the stats dump such that it shows up in the header to the *callgrind_annotate* report under the *Trigger:* listing. The following demonstrates this by listing which of the foo executions triggered the data dump. Notice that the callgrind control statement requires a C string, so we do the little dance between C++ string and C string.

```
#include <valgrind/callgrind.h>
#include <string>

int main(int argc, char* argv[]){

 // code we don't want to profile

 for(int i(0); i<5; i++){

   CALLGRIND_START_INSTRUMENTATION;
   foo();
```

```
    CALLGRIND_STOP_INSTRUMENTATION;
    std::string msg("foo number ");
    msg.append(std::to_string(i));

    CALLGRIND_DUMP_STATS_AT(msg.c_str());
  }

  // more code we're not profiling

}
```

The *to_string* method is a C++11 addition to the *string* library and requires that we compile with *–std=c++11*.