SkyPat: C++ Performance Analysis and Testing Framework

Ping-Hao Chang, Kuan-Hung Kuo, Der-Yu Tsai, Kevin Chen, Luba Tang
Skymizer Software
{peter,ggm,a127a127,kevin,luba}@skymizer.com

Abstract

This paper introduces SkyPat, a C++ performance analysis toolkit on Linux. SkyPat combines unit tests and `perf_event` to give programmers the power of white-box performance analysis.

Unlike traditional tools that manipulate entire program as a black-box, SkyPat works on a region of code like a white-box. It is used as a normal unit test library. It provides macros and assertions, into which `perf_events` are embedded, to ensure correctness and to evaluate performance of a region of code. With `perf_events`, SkyPat can evaluate running time precisely without interference to scheduler. Moreover, `perf_event` also gives SkyPat accurate cycle counts that are useful for tools that are sensitive to variance of timing, such as compilers.

We develop SkyPat under the new BSD license, and it is also the unit-test library of the "bold" project.

1 Introduction

SkyPat is developed by a group of the compiler developers to satisfy compiler developers' needs. From compiler developers' view, correctness and performance evaluation are grand challenges for engineering. Compiler optimizations have no guarantee of performance improvement. Sometimes optimizations degrade performance, and sometimes they introduce new faults in a program. Compiler developers need a tool to verify correctness and performance of each compiler optimizations at the same time.

Traditional tools that evaluate the whole program do not fit our demands. Compilers perform optimization based on knowledge it has to a region of code, such as loops or data flows. We need libraries that can evaluate only a region of code that optimization is interesting in.

For compiler developers, integrating unit-test and performance evaluation libraries for a piece of code is very rational. SkyPat is such library: by using SkyPat, users can evaluate correctness and performance by writing test-cases to evaluate a region of code.

Usually, unit-test tools and performance evaluation tools are separated tools. For example, *GoogleTest* [2] is well-known C++ unit-test framework. *GoogleTest* can evaluate correctness but cannot evaluate performance. Besides, *perf* [1] is well-known performance evaluation toolkit in Linux. *perf* can evaluate performance of programs, including its running time, cycles and so on. Although *perf* can evaluate whole program, using *perf* to evaluate a region of code is difficult.

In this paper, we introduce SkyPat, which combines unit-test and performance evaluation. Programmers only need to write and execute unit-tests and they can get correctness and performance. With the help of `perf_event` of Linux kernel, SkyPat can provide precise timer and additional runtime information to measure a region of code. By integrating unit-test and performance evaluation, SkyPat make evaluation of a region of code easier.

The organization of this paper is as follows. Related work is in Section 2. We present SkyPat's design and implementation in Section 3 and shows SkyPat testing and performance framework in Section 4. At last, we conclude this paper in Section 5.

2 Related Work

Oprofile[3] and *perf* are two most popular performance evaluation tools. Oprofile is capable to evaluate not only the whole system but also a single program. Before versions 0.9.7 and earlier, it was based on sampling-based daemon that collects runtime information. Because sampling-based daemon wasted system resources,

Linux community creates new interfaces, called "Performance Counter"[4] or "`perf_event`". After that, Oprofile is "`perf_event`"-based in the later version.

By the success in "Performance Counter", Linux community builds another performance evaluation tool, called *perf*, based on "Performance Counter", too. *Perf* is a performance evaluation toolkit with no daemon to collect runtime information. *perf* gets runtime information by kernel directly rather than collecting by daemon, therefore, *perf* eliminates lots of overhead to bookkeep profiling information and becomes faster.

Both OProfile and *perf* evaluate performance of the entire program, not a region of code. And of course, it makes some efforts to integrate them with unit-test frameworks.

Regarding unit-test frameworks, *GoogleTest* has become popular and has been adapted by many projects recently. *GoogleTest* is a xUnit test framework for C++ programs. By providing ASSERT and EXPECT macros, *GoogleTest* helps programmers to verify program's correctness by writing test-cases. While executing test-cases, a program stops immediately if it meets a fatal error. If a program meets a non-fatal error, the program shows the runtime value and expected value on the screen.

3 Implementation and Design

SkyPat is a C++ performance analyzing and testing framework on Linux platforms. We refer to the concept of *GoogleTest* and extend its scope from testing framework into Performance Analysis Framework. With the help of SkyPat, programmers who wants to analyze their program, only need to write test cases, and SkyPat tests programs' correctness and performance just like normal unit-test frameworks.

SkyPat provides ASSERT/EXPECT macros for correctness checking and PERFORM macro for performance evaluation. ASSERT is assertion for fatal condition testing and EXPECT is non-fatal assertion. That is to say, if a condition of ASSERT fails, the test fails and stops immediately. On the other hand, when the condition of EXPECT fails, it just shows messages on screen to indicate that is a non-fatal failure and the test keeps going on.

A PERFORM macro is used to arbitrarily wrap an block of code in a testee function. It invokes `perf_events` at the beginning and the end of the block of code to measure the performance. When a program executes at the beginning of the region of code, the PERFORM macro calls a system call to kernel to register a performance monitor to gather process runtime information, such as execution time. When program executes in the end of the region of code, a system call is sent to kernel automatically to disable the monitor. SkyPat calculates the difference of time between the beginning and the end to get the period of runtime information of the region of code.

4 SkyPat Testing and Performance Framework

Here are some examples to show how to use SkyPat to evaluate correctness and performance.

4.1 Declare Test Cases and Test Functions

To create a test, users use the PAT_F() macro to define a test function. A test function can be thought as a ordinary C function without return value. Several similar test functions for the same input data can be grouped as a test case.

Figure 1 shows how to define a test-case and test-functions.

Every PAT_F macro declares a test, with two parameters: test-case and test-function names. In Figure 1, "AircraftCase" is the name of test-case. "take_off_test" and "landing_test" are the names of test-function belongs to "AircraftCase" test case. Test functions grouped in the same test-case are meant to be logically related. Users put ASSERT/EXPECT and PERFORM macros in a test function to evaluate correctness and performance. These macros is described in the following section.

```
PAT_F(AircraftCase, take_off_test)
{
   // Test Code
}

PAT_F(AircraftCase, landing_test)
{
   // Test Code
}
```

Figure 1: Example for declaring a test

```
PAT_F(MyCase, fibonacci_test)
{
  ASSERT_TRUE(0 != fibonacci(10));
  EXPECT_EQ(fibonacci(10), 2);
  ASSERT_NE(fibonacci(10), 3);
}

PAT_F(MyCase, AP_test)
{
  ...
```

Figure 2: Example of assertions

```
[ RUN      ] MyCase.fibonacci_test
[  FAILED  ]
main.cpp:53: error: failed to expect
Value of: 2 == fibonacci(10)
  Actual:   false
  Expected: true
[ RUN      ] MyCase.AP_test
[      OK  ]
```

Figure 3: Output of Figure 2

4.2 Correctness Checking

We copy the concept from *GoogleTest* for our correctness evaluation.

There are several variants of ASSERT macros, ASSERT_TRUE/FALSE, ASSERT_EQ/NE (equal) and ASSERT_GT/GE/LT/TE (great/less) series. If the condition of ASSERT fails, the test will stop and exit the test immediately. EXPECT macros, like ASSERT macros, there are also some similar variants. If the condition of EXPECT fails, the test will not stop but keep execute and display the expected result and real result on screen.

Figure 2 shows how fatal and non-fatal assertions work. "fibonacci" is a testee for illustration. There are three assertions: two fatal and one non-fatal.

Figure 3 shows the output of Figure 2. As we mentioned before, ASSERT assertions stop the execution immediately and EXPECT assertions try to keep the execution going on.

4.3 Performance Evaluation

A PERFORM macro measures the performance of a block of code which it wraps up. Figure 4 shows how to use PERFORM macro.

```
PAT_F(MyCase, fibonacci_perf_test)
{
  PERFORM {
    fibonacci(40);
  }
}
```

Figure 4: Example of PERFORM

```
[ RUN      ] MyCase.fibonacci_test
[CXT SWITCH]       3
[ TIME (ns)]  2363214415
```

Figure 5: Output of Figure 4

The PERFORM macro registers a performance monitor at the beginning of the code which its wraps up and detaches the monitor when leaving the block of code. SkyPat calculates and remembers the performance details whenever detaching a performance monitor.

The result is shown in Figure 5. SkyPat measures the execution time and the number of context-switches during the region of code. It saves efforts at complicated interaction between *perf* and the region of code. Users just use macros, just works like writing a test program, and they can easily get the runtime information of the region of code.

For now, SkyPat measures few information such as the run-time clock cycles. We will add more features, such as cache miss and page faults, in the near future.

4.4 Run All Test Cases

To integrate all test-case, user should call *Initialize* and *RunAll* in their program. `Initialize(&argc, argv)` initializes outputs. `RunAll()` runs all the tests you've declared and prints the results on screen. If any test fails, then `RunAll()` returns non-zero value.

```
int main(int argc, char* argv[])
{
  pat::Test::Initialize(&argc, argv);
  pat::Test::RunAll();
}
```

Figure 6: Example of Initialize and RunAll

5 Conclusion and Future Works

By integrating unit-test framework and performance evaluation tool, user can get correctness and performance metrics for a region of code by writing test-cases. Users can get the performance between the region of code defined by themselves. For programs which needs high precise timing information and other runtime information of the region of code, such as compiler, SkyPat can give them more ability to measure the bottleneck of regions of a program.

References

[1] Arnaldo Carvalho de Melo, Redhat, "The New Linux 'perf' tools" in *17 International Linux System Technology Conference (Linux Kongress), 2010*

[2] GoogleTest, Google, https://code.google.com/p/googletest/

[3] OProfile, http://oprofile.sourceforge.net/about/

[4] Ingo Molnar, "Performance Counters for Linux". *Linux Weekly News, 2009*. http://lwn.net/Articles/337493/