PARASOFT

Build API Performance from the Ground Up:

Using Unit Tests to Benchmark API Component Performance

APIs were originally conceived as basic integration tools that enabled disparate applications to exchange data, but they have evolved into critical glue that binds multiple processes into a single application. Modern applications are aggregating and consuming APIs at a staggering pace in order to achieve business goals. In fact, the business logic of most modern applications now relies on some combination of APIs and third-party libraries—which means that the performance of end-to-end transactions is heavily dependent upon the performance of the APIs and components that the application leverages. Given their ability to make or break performance goals, key application components should undergo a rigorous performance evaluation as an integral part of the acceptance process. The sooner you understand the performance capabilities of an application's key components, the easier it is to remediate problems and the more effectively you can ensure that the integrated application will meet its performance requirements.

Using unit tests to evaluate component performance provides a number of advantages; unit tests:

- Offer a flexible, but at the same time standardized, way of testing at the component level.
- Are well-understood and widely-used in development environments.
- Typically require only a fraction of the hardware resources necessary for testing the entire application. This means you can test the components at maximum projected "Stress" level (see Fig. 1) early and more often with the hardware resources available in development environments.

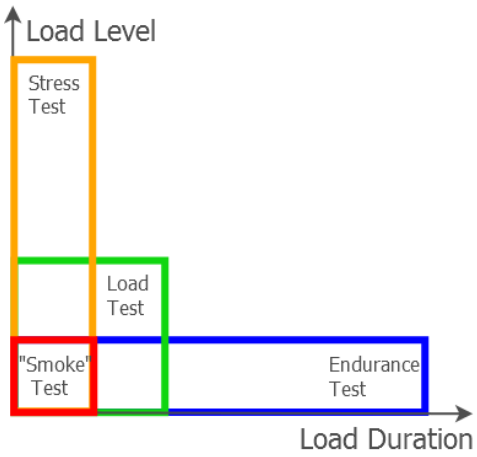

Fig. 1. Approximate load level and test duration proportions of typical performance testing scenarios

However, unit-level performance testing is often overlooked because unit testing tools lack the capabilities commonly found in dedicated performance testing tools (e.g., the ability to set up and execute various performance test configurations, monitoring of system and application resources during the test, collecting and analyzing performance test results, etc.).

This paper explains how you can get the best of both worlds by executing unit-level tests with traditional performance testing tools. It also outlines a strategy you can apply to measure and benchmark the performance of the components that your team might integrate into your target application.

Establishing a Component Benchmarking Workflow

The need for component-level benchmarking in development environments can arise at different stages of the software lifecycle and is driven by the following questions:

- Which of the available third-party components not only satisfies the functional requirements, but also has the best performance? Should I use component A, B, C, etc. or implement a new one? Such questions usually occur at design and prototyping stages.
- Which of the alternative code implementations is the most optimal from the performance perspective? Such questions usually occur during the development stage and are related to code developed internally.

A properly configured and executed component benchmark can help answer these questions. A typical component benchmark workflow (shown in Fig. 2) consists of:

1. Creating unit tests for the benchmarked components.
2. Selecting benchmark performance parameters (these are the same for all components).
3. Executing the performance tests.
4. Comparing the performance profiles of different components.

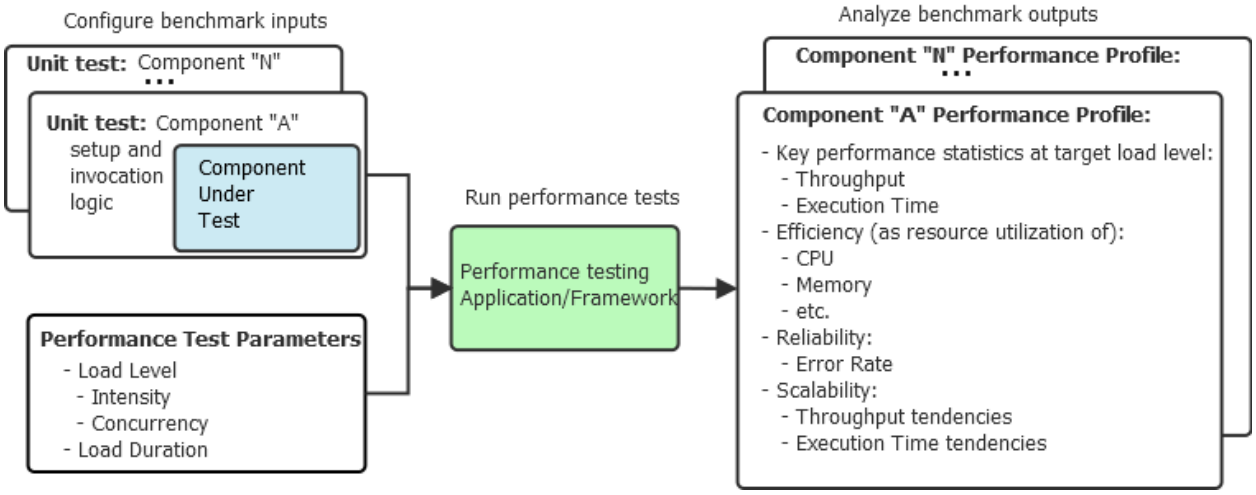

Fig. 2. A typical component benchmark workflow

Introducing a Component Benchmarking Example: JSON Parsers

For a concrete example, let's look at how to compare the performance of four JSON parser components: Jackson (streaming), JsonSmart, Gson, and FlexJson. Here, we will use JUnit as the unit testing framework and Parasoft Load Test, which is part of Parasoft's API Testing solution, as the load testing application. However, the same strategy can be applied with other unit testing frameworks and load testing applications.

Creating Unit Tests for the Benchmarked Components

The JUnit test should invoke the component like the target application would. This applies to the component configuration, the choice of the component API methods, and the values and ranges of method arguments.

The desired level of results verification depends on the nature of the tests. As a rule, you would do more extensive unit test results verification for reliability tests, but perform a more basic level of verification for efficiency tests (since 'heavy' results verification logic may skew the performance picture). On the other hand, it is important to perform at least some verification to ensure that the component gets properly configured and invoked.

In our benchmark, the JSON content is loaded from a file at the beginning of the performance test and cached in memory. The JSON file has a size of 225KB and contains 215 top-level account description objects. Each account object has a "balance" name/value pair:

```
{
    "id": "54b98c2b7b3bd64aae699040",
    "index": 214,
    "guid": "565c44b0-9e6d-4b8e-819c-48aa4dd9d7c2",
    "balance": "$3,809.46",

...

}
```

The basic level of component functionality verification is implemented in the following way: the JUnit test invokes the parser API to find all the "balance" elements inside the JSON content and calculate the total balance of all account objects in the JSON document. The calculated balance is then compared with the expected value:

```
public class JacksonStreamParserTest extends TestCase {
    @Test
    public void testParser() throws IOException {
        float balance = 0;
        JsonFactory jsonFactory = new JsonFactory();
        String json = JsonContentProvider.getJsonContent();
        JsonParser jp = jsonFactory.createJsonParser(json);
        while (jp.nextToken() != null) {
            String fieldname = jp.getCurrentName();
            if (fieldname != null && fieldname.equals("balance")) {
                jp.nextToken();
                balance += TestUtil.parseCurrencyStr(jp.getText());
            }
        }
        TestUtil.assertBalance(balance);
    }
}
```

Because we are comparing parsers for efficiency, we are using 'lightweight' resource verification to avoid distorting the performance picture. If you choose to do some 'heavyweight' results verification,

you can compensate for it in the same way that we compensate for the resource consumption of the performance testing framework (described below.) However, this will require a more elaborate preparation of the Baseline Scenario.

Compensating for the Testing Framework

Because we are comparing the performance of components that run in the same process as the container performance testing application, we need to separate the share of the system-level and application-level resources consumed by this application and the JUnit framework from the share consumed by the component itself. To estimate this share, we can run a benchmark load test scenario with an empty JUnit test:

```
public class BaselineTest extends TestCase {
    @Test
    public void testBaseline() {
    }
}
```

If the resource utilization levels of the baseline scenario are not negligible, we need to subtract these levels from the levels of the component benchmark runs to compensate for the share of resources consumed by the testing framework.

Selecting and Configuring Benchmark Performance Parameters

The test environment setup should emulate the major parameters of the target environment, such as the operating system, the JVM version, JVM options such as the GC options, the server mode, and so forth. It may not always be possible or practical to reproduce all the deployment parameters in the test environment; however, the closer it is, the less of chance that the component performance during the test will differ from that on the target environment.

Understanding Concurrency, Intensity, and Test Duration

The major performance test parameters that determine the conditions under which the components will be tested are load level (defined as load intensity and load concurrency) and load duration (see Fig. 3).

To shape these generic performance test parameters into concrete forms, start by examining the performance requirements of the target application where you anticipate using this component. Application-level performance requirements can provide concrete or approximate characteristics of the load level to which the component should be subjected. Translating application performance requirements into component performance requirements, however, presents multiple challenges. For example, if the application performance requirement states that it must respond within M milliseconds at a load level of N users or requests per second, how does this translate to performance requirements for a component that is a part of this application? How many requests to a specific component will one request to the application generate?

If there is an older version of the application, you can make an educated guess by tracing a few application-level calls or by examining call trace statistics collected by an APM (Application Performance Management) tool. If neither of these options are available, the answer can come from examining the application design.

If the component load parameters cannot be deduced from the target application performance specifications or if such a specification is not available, then an alternative choice for a benchmark is to run at the maximum load level that can be achieved on the hardware available for the test. However, the risk involved in this approach, when taken without any regard for the expected load levels, is that the benchmark results may not be relevant in the context of the target application.

In any case—but particularly for the benchmarks where the load levels are driven by the performance limits of the hardware available to the test—be aware of how the resource overutilization impacts the test results. For example, when comparing component execution times, it is generally advisable to stay below the 75-80% CPU utilization level. Increasing the utilization above that level can adversely affect the accuracy of test execution time measurement and may distort the benchmark results.

Often, the aggregate performance testing parameter 'load level' is not explicitly separated into its major parts: intensity and concurrency (see Fig. 3). This can lead to an inadequate performance test configuration.

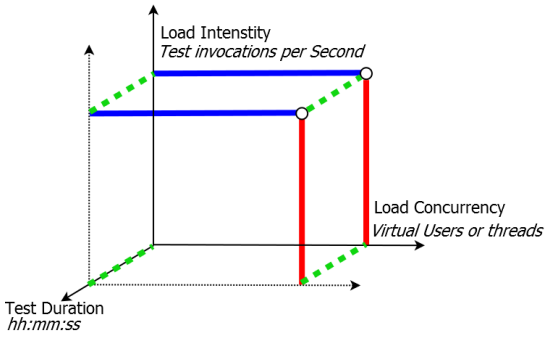

Fig. 3. The major performance test parameters: intensity, concurrency, and duration

Load Intensity is the rate of requests or test invocations at which the component will be tested. In a performance testing application, the load intensity can be configured directly by setting the hit/test per second/minute/hour levels of a performance test scenario. It can also be configured indirectly as a combination of the number of virtual users and the virtual user think time.

Load Concurrency (in our context) can be described as the degree of parallelism with which a load of a given intensity is applied. Concurrency level can be configured by the number of virtual users or threads in a load test scenario. Setting an appropriate concurrency level is essential when testing for potential race conditions and for performance problems due to thread contention and access to shared resources—as well as for measuring the memory footprint of multiple instances of the component.

Test Duration for a component benchmark test depends on the test goals. When approached from the mathematical standpoint, one of the major factors in determining the test duration is the statistical significance of the load test data set. However, this type of approach may be too complicated for everyday practical purposes. The following 'rules of thumb' are generally sufficient:

- **When comparing relative resource utilization of components**, use the ramp up section of the performance scenario to gradually bring the system to the target load level. The test can be stopped after the parameters used to compare performance have stabilized.
- **When testing for reliability**, use the duration of the endurance test scenario (from your application's performance test requirements document) as a starting point. The diagram in Fig. 1 illustrates approximate "load level to test duration" proportions of typical application performance testing scenario types. The long-running tests, such as endurance (or “soak”) tests, are used to uncover failures caused by slow-building resource leaks— such as application memory leaks or a build-up of log files on disk—and to capture rare errors, such as those caused by race conditions.

Once the concrete performance test parameters have been established, you can use them to configure the load test application. After you run the performance tests with different components, you can start analyzing the benchmark outputs.

Configuring Concurrency, Intensity, and Test Duration

Assume that we examined the target application's performance requirements and found that the maximum projected load level for the application is 1000 concurrent users, the maximum request rate is 1000 requests per second, and the maximum expected JSON request size is 200KB. Each request to the target application translates to one call to the JSON parsing component.

Based on these projections, we construct the following performance test scenario:

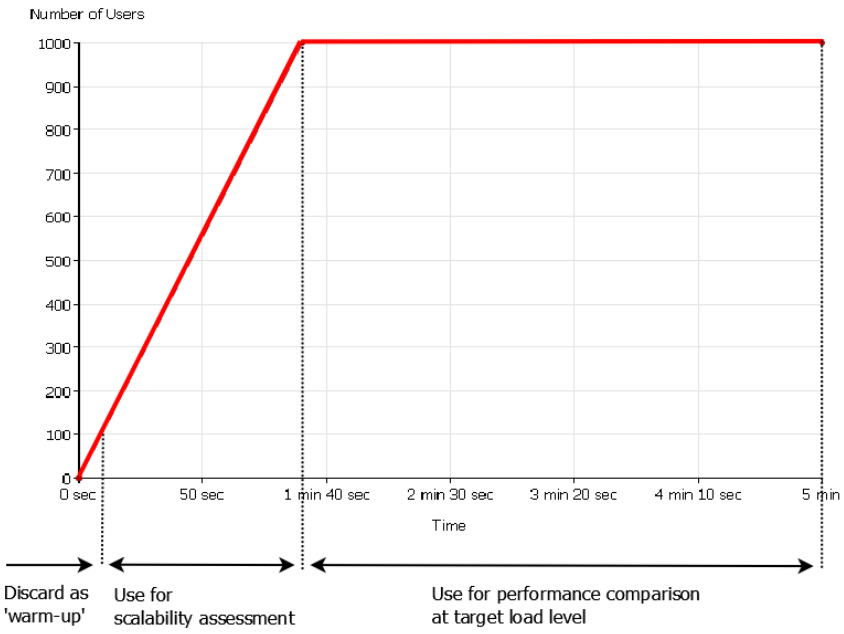

Fig. 4. Performance testing scenario for the benchmark

The horizontal ‘steady load’ section of the scenario will be used to compare performance at the maximum projected (target) load level. The ‘linear increase’ section will gradually bring the load to the

target level. We will also use this section for a preliminary component scalability assessment, since this section can show us some dynamics of the component's response to the increasing load. The start-up section of the scenario will be discarded as a stage for the performance testing application's resource loading and the 'warm up' of the JVM JIT (just in time) compiler. We expect that the length of the horizontal section of the scenario is sufficient for the system parameters to stabilize (you can run some preliminary tests to see how long it takes).

The following metrics will be used to construct the performance profiles of the parser components:

- Ability to reach the desired test per second (TPS) rate on the selected hardware.
- Average system CPU utilization.
- Memory consumption measured as the JVM free memory.
- Average test execution time (as recorded by the performance testing tool).
- Percentage of errors (as recorded by the performance testing tool).

Executing the Performance Tests

Each performance test of the benchmark is executed in a new process to prevent the JVM and the application state at the end of the current test from affecting the following test.

Analyzing Benchmark Outputs

You can use the load test application's benchmark run data to create *component performance profiles*. We define a component performance profile as a set of benchmark-relevant metrics that describe the component's response to the applied load. The most common metrics categories in a performance profile are key performance statistics, efficiency, reliability and scalability.

- **Key performance statistics** includes the response time (measured as the unit test execution time) and the throughput (measured as test execution rate).
- **Efficiency** includes metrics that measure the component's use of system-level and application-level resources at the target load level.
- **Reliability** includes metrics that measure the error rates for certain load levels and durations. It is important to remember that the causes of errors during a load test can be different from those in a single unit test invocation. For example, load-test-specific errors can occur due to component thread safety issues, memory leaks, or failure to release system-level resources (sockets, file handles) or application-level resources (database connections).
- **Scalability** includes metrics that measure the component's ability to utilize system-level and application-level resources to handle the growing amount of work. For components designed to work in multithreaded environments, scalability can be evaluated as the ability to increase throughput by utilizing a growing number of threads. By evaluating component scalability tendencies, we can (to a certain degree) anticipate component performance at load levels outside of those used in the test.

A component performance profile may include all or some of the above listed metrics, depending on the benchmark goals. It may not always be possible to create a single benchmark test that evaluates the component performance from all desired angles; for such cases, a set of tests should be created.

Findings: Performance at Target Load Levels

In our example, the reports' 'steady load' sections were separated from the rest of the reports for comparing performance at target load levels. Fig. 5 shows the dynamic CPU utilization levels during the test.

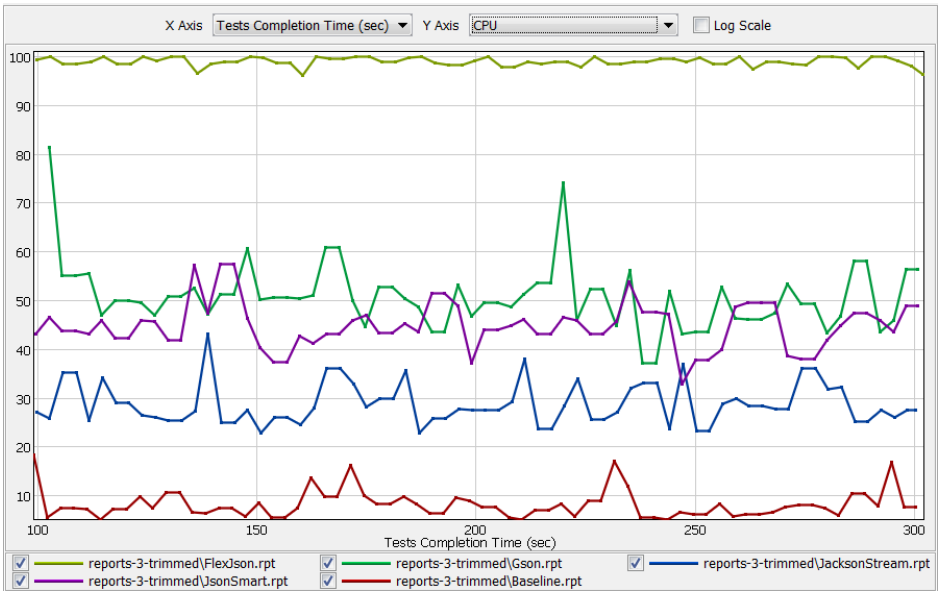

Fig. 5. CPU utilizations of the benchmarked parsers at target load level[1]

Fig. 6 presents the performance profiles of the benchmarked components. The table contains values of aggregate statistics automatically calculated by the performance testing tool (highlighted in green) as well as values derived from the automatically-calculated metrics (highlighted in blue).

	Baseline	JacksonStream	JsonSmart	Gson	FlexJson
Reached the target 1000 TPS load level	N/A	Yes	Yes	Yes	No
Average CPU Utilization (%)	8.2	28.8	44.9	50.8	99
Average JMV free memory (MB)	1010	838	777	771	838
Average test execution time (ms)	0	1	3	3	17
Error Rate (%)	0	0	0	0	0
Baseline-compensated average CPU Utilization (%)	N/A	20.6	36.7	42.6	99[2]
Baseline-compensated Relative Throughput (HPS/CPU%)	N/A	48.5	27.2	23.5	5.1[2]

Fig. 6. Performance profiles of the benchmarked components.

[1] These test results are not intended to make any generalizations about the performance of the selected parsers, but rather to serve a component benchmark demonstration based on the selected JSON content and parser invocation patterns specific to this example.
[2] Unchanged; the test did not achieve the target execution rate.

The derived metrics are calculated as follows:

- **Baseline-compensated Average CPU Utilization (%)** is the difference between the average CPU utilization of the baseline scenario (8.2%) and the average CPU utilization of each of the component tests.
- **Baseline-compensated Relative Throughput (HPS / CPU%)** is the hit per second rate per one percent of CPU utilization.

The baseline-compensated metrics provide a more accurate comparison of component performance since they factor out the testing framework's share of resource consumption. The Relative Throughput metric was added because a test in the benchmark (FlexJson) approached 100% CPU utilization without reaching the target 1000 TPS level.

By revealing that a stream parser has a significant performance advantage over the object-based parsers for the selected test configuration, the benchmark results help us select which component to use.

Findings – Component Scalability Assessment

The linear increase section of the reports can be used to assess component scalability on the interval from 100 to 1000 TPS as its ability to handle the growing amount of work. Fig. 7 shows the maximum (yellow) and average (gray) execution times of the JsonSmart parser component plotted along the vertical axis as a function of the TPS rate plotted along the horizontal axis ('Tests Completion Rate (1/sec)' in the graph).

The graphs show that on the interval of 100 to 1000 TPS, the maximum and average component execution times do not significantly change as a function of the increasing load. A linear—or worse—exponential increase of the test execution time as a function of the increasing test execution rate would be a concern from a scalability perspective, indicating a potential bottleneck in the component.

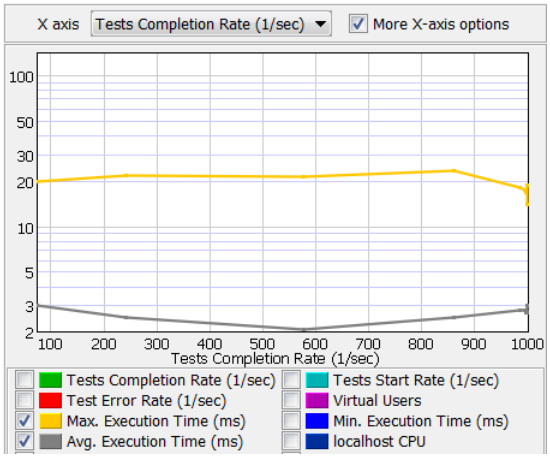

Fig. 7. Component maximum (yellow) and average (gray) execution times as a function of the test invocation rate

To evaluate component scalability as the ability to utilize threads to increase throughput, we should run a performance test with no Virtual User think time. The Throughput to Thread Count report graph of such a performance test run of the JsonSmart component is presented in Fig. 8. Because of the test

configuration, the number of threads executing the component code is equal to the number of Virtual Users; for this reason, we interpret the Virtual Users plotted along the horizontal line as threads. The graph shows close-to-linear component scalability with the maximum throughput observed at 8 threads, which is equal to the number of the CPU cores on the test machine.

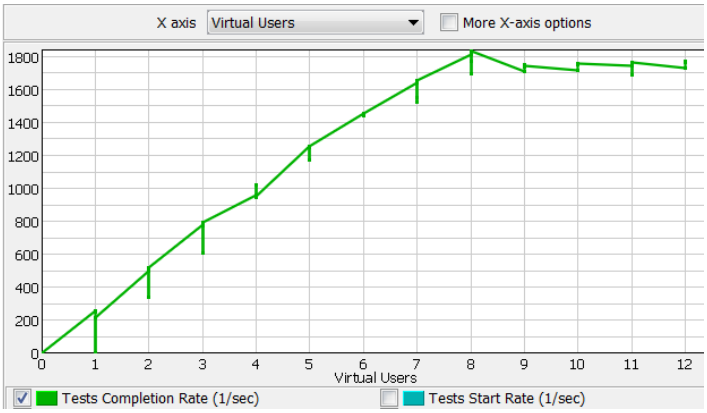

Fig. 8. A Throughput to Thread Count graph of a JsonSmart component

An example of a Throughput to Thread Count graph showing poor scalability of a component that uses a synchronized implementation of java.util.Map is shown in Fig. 9. On an 8 CPU core machine, the maximum throughput is observed at 2 threads (green graph) and decreases after more threads are added. The average test execution time is growing linearly as a function of the number of threads (gray graph).

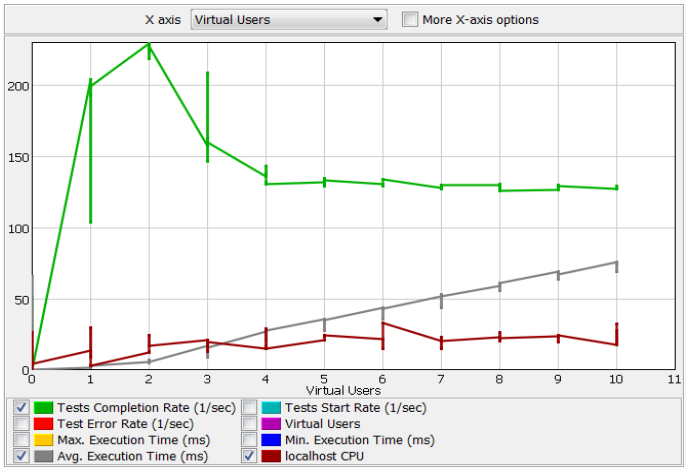

Fig. 9. A Throughput to Thread Count graph of a component with poor scalability

Additional Steps

The same benchmarking methodology described above can be used to compare the performance of different component configurations and alternative methods of invoking the component API. After

selecting a component with the best performance profile and finding the best (from the performance perspective) way of using the component, consider creating a façade to the component API that configures and invokes it in an optimal way. Creating such a façade will help ensure that other developers use the component in the most efficient manner.

Conclusion

Component-level benchmarking facilitates optimal decision-making related to application performance throughout the software development lifecycle. When applied systematically in development organizations, this type of testing will improve developers' understanding of the performance implications of various programming patterns, eliminate the guesswork related to code performance, and help establish a culture where software performance is as much a concern as the software functionality. The use of a common method (elements of which were described in this paper), a common testing standard (such as JUnit), and a common performance testing tool can help organizations implement the practice of component benchmarking in a systematic way.

About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. To combat the risk of software failure while accelerating the SDLC, Parasoft offers a Development Testing Platform and Continuous Testing Platform. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, API testing, dev/test environment management, service virtualization and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

Contacting Parasoft

Headquarters

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Email: info@parasoft.com

Global Offices

Visit www.parasoft.com/contact for contacting Parasoft in EMEAI, APAC, and LATAM.

Author Information

This paper was written by Sergei Baranov, Principal Software Engineer at Parasoft