Performance

Analysis of performance with benchmarks

Introduction

Lumberjack was designed from the start to be fast. And fast it is! The project itself comes with a suite of benchmarking tests. With a simple build-and-go in Xcode you can run them yourself. But the main reason the benchmarking tests are there is so the project maintainers never neglect the performance factor.

But what makes lumberjack so fast?

Grand Central Dispatch

Lumberjack takes advantage of grand central dispatch (GCD) if it is available on the target platform. For example, the framework will execute each individual logger concurrently. This means that the framework can write log messages to the console and a file at the same time, using multiple CPU cores. And GCD optimizes all this multi-threading for us automatically, meaning the framework can scale from 1 CPU core to 100.

If GCD is not available, Lumberjack still performs excellently using traditional multi-threaded optimizations.

Asynchronous Logging

Lumberjack has the ability to execute certain log statements asynchronously. (This is, of course, optional and you can disable it or fine-tune it as you see fit.)

The default philosophy for asynchronous logging is very simple:

- Log messages with errors should be executed synchronously.

> After all, an error just occurred. The application could be unstable.

- All other log messages, such as debug output, are executed asynchronously.

> After all, if it wasn't an error, then it was just informational output, or something the application was easily able to recover from.

This is easily configurable. And the performance of the library does not solely rely on asynchronous logging.

A Better NSLog

The simple truth is that NSLog is just plain slow.

But why? To answer that question, let's find out what NSLog does, and then how it does it.

What does NSLog do exactly?

NSLog does 2 things:

1. It writes log messages to the Apple System Logging (asl) facility. This allows log messages to show up in Console.app.
2. It also checks to see if the application's stderr stream is going to a terminal (such as when the application is being run via Xcode). If so it writes the log message to stderr (so that it shows up in the Xcode console).

Writing to STDERR doesn't sound difficult. That can be accomplished with fprintf and the stderr file descriptor reference. But what about asl?

The best documentation I've found about ASL is a 10 part blog post from Peter Hosey: link

Without going into too much detail, the highlight (as it concerns performance) is this:

To send a log message to the ASL facility, you basically open a client connection to the ASL daemon and send the message. BUT - each thread must use a separate client connection. So, to be thread safe, every time NSLog is called it opens a new asl client connection, sends the message, and then closes the connection. (1) The lumberjack framework avoids much of the cost by creating a single re-usable asl client connection for its background logging thread.

1 - Assuming that NSLog acts like its open-source cousin CFShow.

Benchmarking

As mentioned earlier, the Lumberjack framework comes with a suite of benchmarking tests. You can run these benchmark tests yourself using the "BenchmarkMac" or "BenchmarkIPhone" Xcode projects. When you build-and-go, the project will output the results to the Xcode console in a human-readable format at the end of the benchmarking process. It will also output a CSV file in case you wanted to graph the results.

There are 4 main tests. The base case for each test is a standard NSLog statement. Each test is run 20 times, from which a min, max and average time is calculated. The benchmark includes various configurations of the Lumberjack framework, such as:

- logging to the console only
- logging to a file only
- logging to the console & file at the same time

Test # 1:

> Asynchronous logging test. Execute 1,000 log statements. All lumberjack log statements are queued onto a background logging thread.

Test #2:

Synchronous logging test. Execute 1,000 log statements. All lumberjack log statements are executed synchronously. They may still be executed on a background logging thread, but the original log statement does not return until the logging has completed.

Test #3:

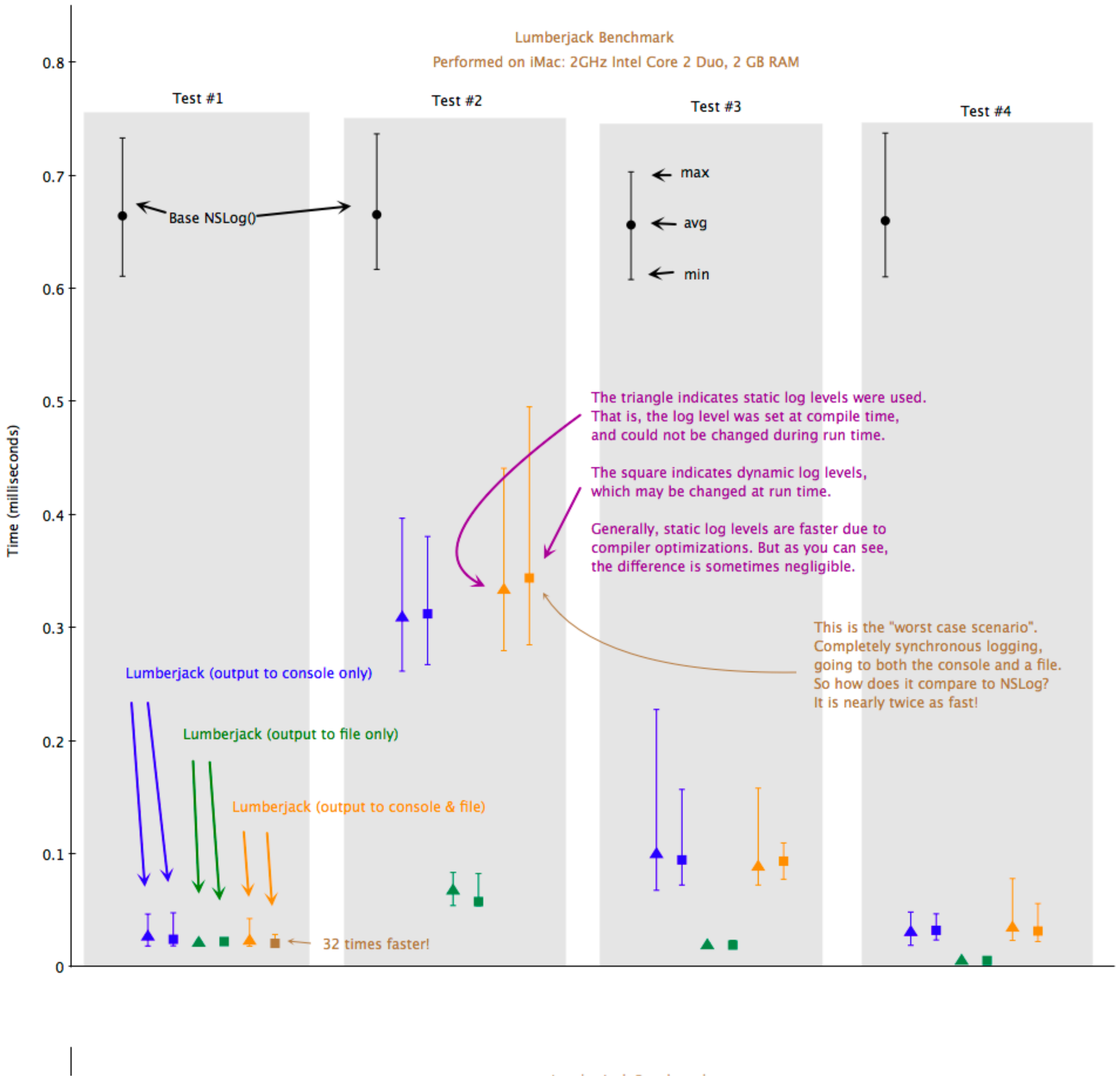
Real life simulation with an even spread of log statements across various levels. Execute 1,000 log statements. 25% will be above the log level and will be filtered out. 50% will be executed asynchronously. 25% will be executed synchronously.

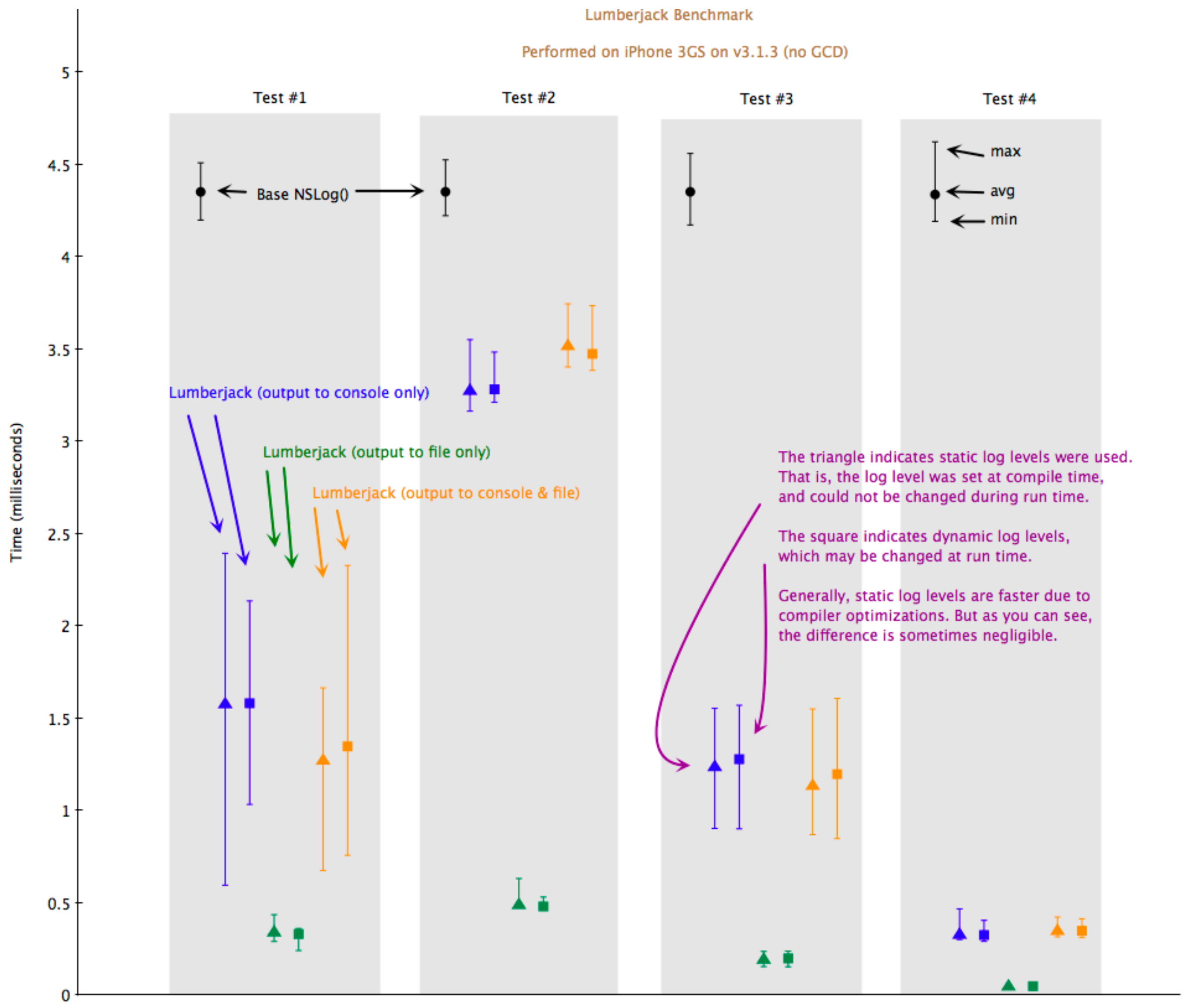
Test #4:

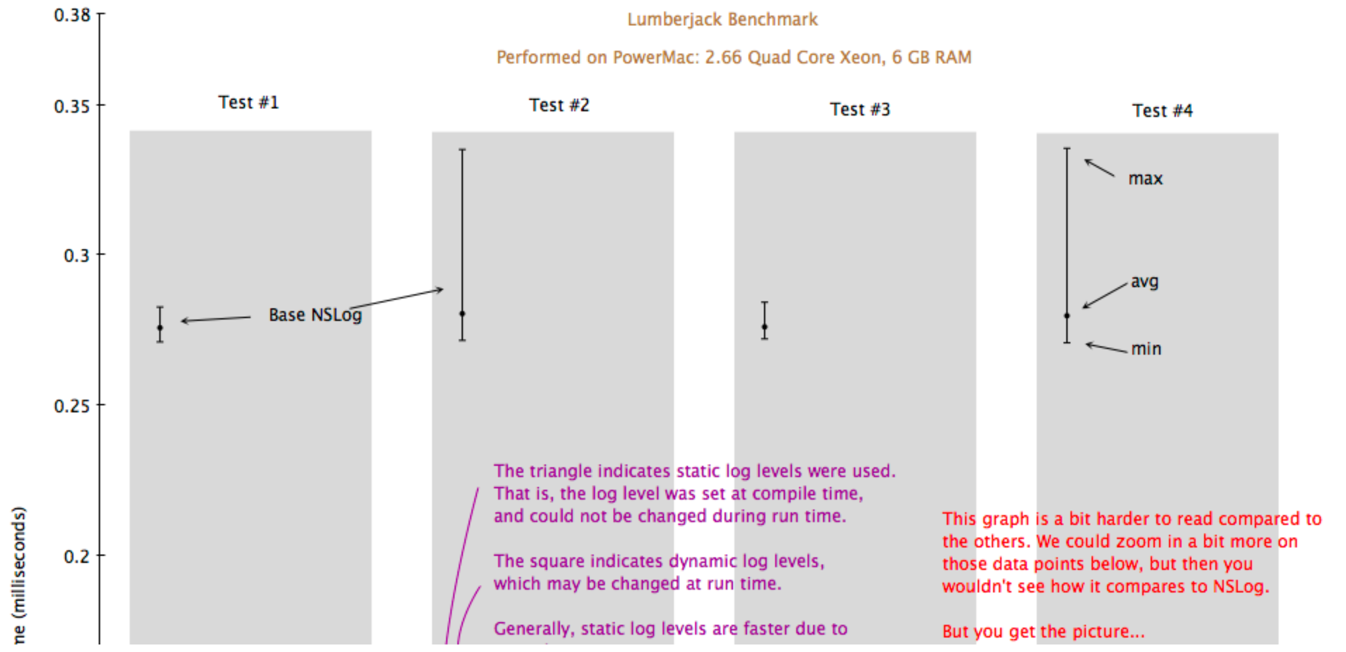
Real life simulation with an abundance of debug log statements. Execute 1,000 log statements. 90% will be filtered out. 10% will be error messages, executed synchronously.

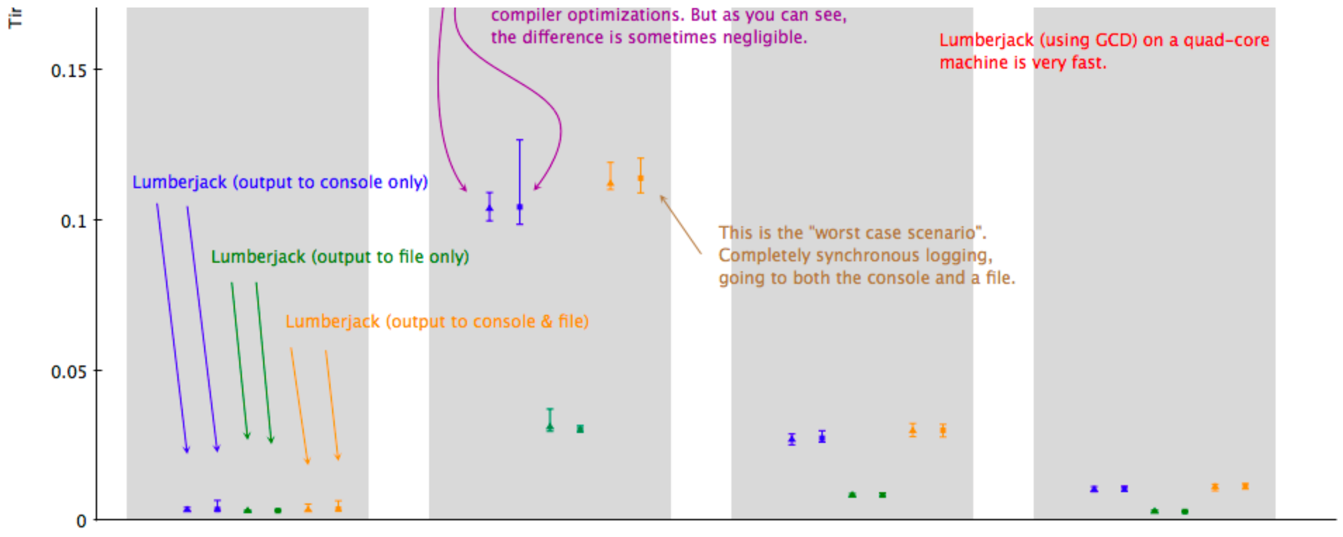
The numbers for test #4 came from some projects that I've worked on in the past. A bit of regex revealed that 90% of my log statements were debug messages. The other 10% were spread among the various other levels, but I wanted the benchmark test to be "worst case" with respect to this 10%.

Note that all of these benchmark test are configurable. So if you wanted to try, for example, test #4 with different percentages, you can easily change the test yourself.




Lumberjack Benchmark
Performed on iPhone 3GS on v3.1.3 (no GCD)
Time (milliseconds)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Test #1
Test #2
Test #3
Test #4
Base NSLog()
max
avg
min
Lumberjack (output to console only)
Lumberjack (output to file only)
Lumberjack (output to console & file)
The triangle indicates static log levels were used. That is, the log level was set at compile time, and could not be changed during run time.
The square indicates dynamic log levels, which may be changed at run time.
Generally, static log levels are faster due to compiler optimizations. But as you can see, the difference is sometimes negligible.


Lumberjack Benchmark
Performed on PowerMac: 2.66 Quad Core Xeon, 6 GB RAM
ne (milliseconds)
0.2
0.25
0.3
0.35
0.38
Test #1
Test #2
Test #3
Test #4
Base NSLog
max
avg
min
The triangle indicates static log levels were used. That is, the log level was set at compile time, and could not be changed during run time.
The square indicates dynamic log levels, which may be changed at run time.
Generally, static log levels are faster due to
This graph is a bit harder to read compared to the others. We could zoom in a bit more on those data points below, but then you wouldn't see how it compares to NSLog.
But you get the picture...



Comment by robbieha...@voalte.com, Nov 01, 2010

The Y-axis should read "Time (seconds)" and NOT "Time (milliseconds)"