Swift Benchmark Suite

This directory contains the Swift Benchmark Suite.

Running Swift Benchmarks

To run Swift benchmarks, pass the `--benchmark` flag to `build-script`. The current benchmark results will be compared to the previous run's results if available. Results for each benchmark run are logged for future comparison.

For branch based development, take a baseline benchmark on the Swift `main` branch, switch to a development branch containing potentially performance impacting changes, and run the benchmarks again. Upon benchmark completion, the benchmark results for the development branch will be compared to the most recent benchmark results for `main`.

Building the Swift Benchmarks

The swift benchmark suite currently supports building with CMake and SwiftPM. We support the following platforms respectively.

- CMake: macOS, iOS, tvOS, watchOS
- SwiftPM: macOS, linux

We describe how to build both standalone and with build-script below.

build-script invoking CMake

By default, Swift benchmarks for OS X are compiled during the Swift build process. To build Swift benchmarks for additional platforms, pass the following flags:

```
$ swift/utils/build-script --ios --watchos --tvos
```

OS X benchmark driver binaries are placed in `bin` alongside `swiftc`. Additional platform binaries are placed in the `benchmark/bin` build directory.

The required Swift standard library dylibs are placed in `lib`. The drivers dynamically link Swift standard library dylibs from a path relative to their run-time location (../lib/swift) so the standard library should be distributed alongside them.

CMake Standalone (no build-script)

To build the Swift benchmarks using only an Xcode installation: install an Xcode version with Swift support, install cmake 3.19.6 or higher, and ensure Xcode is selected with xcode-select.

The following build options are available:

- `-DSWIFT_EXEC`
  - An absolute path to the Swift driver (`swiftc`) to use to compile the benchmarks (default: Xcode's `swiftc`)
- `-DSWIFT_LIBRARY_PATH`
  - An absolute path to the Swift standard library to use during compilation (default: `swiftc_directory`/../lib/swift)
- `-DSWIFT_DARWIN_XCRUN_TOOLCHAIN`

    - The Xcode toolchain to use when invoking `xcrun` to find `clang`. (default: XcodeDefault)
- `-DONLY_PLATFORMS`
    - A list of platforms to build the benchmarks for (default: "macosx;iphoneos;appletvos;watchos")
- `-DSWIFT_OPTIMIZATION_LEVELS`
    - A list of Swift optimization levels to build against (default: "O;Onone;Osize")
- `-DSWIFT_BENCHMARK_USE_OS_LIBRARIES`
    - Enable this option to link the benchmark binaries against the target machine's Swift standard library and runtime installed with the OS. (default: OFF)
- `-DSWIFT_BENCHMARK_GENERATE_DEBUG_INFO`
    - Enable this option to compile benchmark binaries with debug info. (default: ON)

The following build targets are available:

- `swift-benchmark-macosx-x86_64`
- `swift-benchmark-macosx-arm64`
- `swift-benchmark-iphoneos-arm64e`
- `swift-benchmark-iphoneos-arm64`
- `swift-benchmark-iphoneos-armv7`
- `swift-benchmark-appletvos-arm64`
- `swift-benchmark-watchos-armv7k`

Build steps (with example options):

1. `$ mkdir build; cd build`
2. `$ cmake [path to swift src]/benchmark -G Ninja -DSWIFT_EXEC=[path to built swiftc]`
3. `$ ninja swift-benchmark-macosx-$(uname -m)`

Benchmark binaries are placed in `bin`.

The binaries dynamically link Swift standard library dylibs from a path determined by the configuration. If `SWIFT_LIBRARY_PATH` is set, they link against the absolute path provided, regardless of where the binaries are installed. Otherwise, the runtime library path is relative to the benchmark binary at the time it was executed (`@executable_path/../lib/swift/<platform>`).

For example, to benchmark against a locally built `swiftc`, including any standard library changes in that build, you might configure using:

```
cmake <src>/benchmark -G Ninja -DSWIFT_EXEC=<build>/swift-macosx-$(uname -m)/bin/swiftc
ninja swift-benchmark-iphoneos-arm64
```

To build against the installed Xcode, simply omit SWIFT_EXEC:

```
cmake <src>/benchmark -G Ninja
ninja swift-benchmark-iphoneos-arm64
```

In both examples above, to run the benchmarks on a device, the dynamic libraries must then be copied onto the device into the library path relative to `swiftc`. To benchmark against the target machine's installed libraries instead, enable `SWIFT_BENCHMARK_USE_OS_LIBRARIES`.

```
cmake <src>/benchmark -G Ninja -DSWIFT_BENCHMARK_USE_OS_LIBRARIES=ON
ninja swift-benchmark-iphoneos-arm64
```

This will reflect the performance of the Swift standard library installed on the device, not the one included in the Swift root.

build-script using SwiftPM+LLBuild

To build the benchmarks using build-script/swiftpm, one must build both swiftpm/llbuild as part of one's build and create a "just-built" toolchain. This toolchain is then used by build-script to compile the benchmarks. This is accomplished by passing to build-script the following options:

```
swift-source$ swift/utils/build-script --swiftpm --llbuild --install-swift --install-swiftpm --install-llbuild --toolchain-benchmarks
```

build-script will then compile the toolchain and then build the benchmarks 3 times, once for each optimization level, at the path `./build/benchmarks-$PLATFORM-$ARCH/bin/Benchmark_$OPT`:

Standalone SwiftPM/LLBuild

The benchmark suite can be built with swiftpm/llbuild without needing any help from build-script by invoking swift build in the benchmark directory:

```
swift-source/swift/benchmark$ swift build --configuration release
swift-source/swift/benchmark$ .build/release/SwiftBench
#,TEST,SAMPLES,MIN(μs),MAX(μs),MEAN(μs),SD(μs),MEDIAN(μs)
1,Ackermann,1,169,169,169,0,169
2,AngryPhonebook,1,2044,2044,2044,0,2044
...
```

Editing in Xcode

It is now possible to work on swiftpm benchmarks in Xcode! This is done by using the ability swiftpm build of the benchmarks to generate an xcodeproject. This is done by running the commands:

```
swift-source/swift/benchmark$ swift package generate-xcodeproj
generated: ./swiftbench.xcodeproj
swift-source/swift/benchmark$ open swiftbench.xcodeproj
```

Assuming that Xcode is installed on ones system, this will open the project in Xcode. The benchmark binary is built by the target 'SwiftBench'.

NOTE: Files added to the Xcode project will not be persisted back to the package! To add new benchmarks follow the instructions from the section below!

NOTE: By default if one just builds/runs the benchmarks in Xcode, the benchmarks will be compiled with -Onone!

Using the Benchmark Driver

Usage

`./Driver [ test_name [ test_name ] ] [ option [ option ] ]`

- `--num-iters`
  - Control the number of loop iterations in each test sample
- `--num-samples`
  - Control the number of samples to take for each test
- `--list`
  - Print a list of available tests matching specified criteria
- `--tags`
  - Run tests that are labeled with specified tags (comma separated list); multiple tags are interpreted as logical AND, i.e. run only test that are labeled with all the supplied tags
- `--skip-tags`
  - Don't run tests that are labeled with any of the specified tags (comma separated list); default value: `skip,unstable`; to get complete list of tests, specify empty `--skip-tags=`

Examples

- `$ ./Benchmark_O --num-iters=1 --num-samples=1`
- `$ ./Benchmark_Onone --list`
- `$ ./Benchmark_Osize Ackermann`
- `$ ./Benchmark_O --tags=Dictionary`
- `$ ./Benchmark_O --skip-tags=unstable,skip,validation`

Deterministic Hashing Requirement

To run benchmarks, you'll need to disable randomized hash seeding by setting the `SWIFT_DETERMINISTIC_HASHING` environment variable to `1`. (You only need to do this when running the benchmark executables directly -- the driver script does this for you automatically.)

- `$ env SWIFT_DETERMINISTIC_HASHING=1 ./Benchmark_O --num-iters=1 --num-samples=1`

This makes for more stable results, by preventing random hash collision changes from affecting benchmark measurements. Benchmark measurements start by checking that deterministic hashing is enabled and they fail with a runtime trap when it isn't.

If for some reason you want to run the benchmarks using standard randomized hashing, you can disable this check by passing the `--allow-nondeterministic-hashing` option to the executable.

- `$ ./Benchmark_O --num-iters=1 --num-samples=1 --allow-nondeterministic-hashing`

This will affect the reliability of measurements, so this is not recommended.

Benchmarking by Numbers

As a shortcut, you can also refer to benchmarks by their ordinal numbers. These are printed out together with benchmark names and tags using the `--list` parameter. For a complete list of all available performance tests run

- `$ ./Benchmark_O --list --skip-tags=`

You can use test numbers instead of test names like this:

- `$ ./Benchmark_O 1 42`
- `$ ./Benchmark_Driver run 1 42`

Test numbers are not stable in the long run, adding and removing tests from the benchmark suite will reorder them, but they are stable for a given build.

Using the Harness Generator

`scripts/generate_harness/generate_harness.py` runs `gyb` to automate generation of some benchmarks.

** FIXME ** `gyb` should be invoked automatically during the build so that manually invoking `generate_harness.py` is not required.

Adding New Benchmarks

Adding a new benchmark requires some boilerplate updates. To ease this (and document the behavior), a harness generator script is provided for both single/multiple file tests.

To add a new single file test, execute the following script with the new of the benchmark:

```
swift-source$ ./swift/benchmark/scripts/create_benchmark.py YourTestNameHere
```

The script will automatically:

1. Add a new Swift file (`YourTestNameHere.swift`), built according to the template below, to the `single-source` directory.
2. Add the filename of the new Swift file to `CMakeLists.txt` .
3. Edit `main.swift` by importing and registering your new Swift module.

No changes are needed to the Package.swift file since the benchmark's Package.swift is set to dynamically lookup each Swift file in `single-source` and translate each of those individual .swift files into individual modules. So the new test file will be automatically found.

To add a new multiple file test:

1. Add a new directory and files under the `multi-source` directory as specified below:

   ```
   +-- multi-source
   |   +-- YourTestName
   |   |   +-- TestFile1.swift
   |   |   +-- TestFile2.swift
   |   |   +-- TestFile3.swift
   ```

   At least one file must define a public `YourTestName` variable, initialized to an instance of BenchmarkInfo (specified in the template below).

2. In `CMakeLists.txt` add the new directory name to `SWIFT_MULTISOURCE_SWIFT_BENCHES` , and set `YourTestName_sources` to the list of source file paths.

3. Edit `main.swift` . Import and register your new Swift module.

No changes are needed to the swiftpm build since it knows how to infer multi-source libraries automatically from the library structure.

Note:

The benchmark harness will execute the routine referenced by `BenchmarkInfo.runFunction`.

The benchmark driver will measure the time taken for `N = 1` and automatically calculate the necessary number of iterations `N` to run each benchmark in approximately one second, so the test should ideally run in a few milliseconds for `N = 1`. If the test contains any setup code before the loop, ensure the time spent on setup is insignificant compared to the time spent inside the loop (for `N = 1`) -- otherwise the automatic calculation of `N` might be significantly off and any performance gains/regressions will be masked by the fixed setup time. If needed you can multiply N by a fixed amount (e.g. `1...100*N`) to achieve this.

Performance Test Template

```
// YourTestName benchmark
//
// rdar://problem/00000000
import TestsUtils

public let benchmarks = [
  BenchmarkInfo(
    name: "YourTestName",
    runFunction: run_YourTestName,
    tags: [.regression])
]

@inline(never)
public func run_YourTestName(n: Int) {
    # Declare variables

    for i in 1...n {
        # Perform work

        # Verify work was done; break otherwise
    }

    # Assert with CheckResults that work was done
}
```

The current set of tags are defined by the `BenchmarkCategory` enum in `TestsUtils.swift` .

Testing the Benchmark Drivers

When working on tests, after the initial build

```
swift-source$ ./swift/utils/build-script -R -B
```

you can rebuild just the benchmarks:

```
swift-source$ export SWIFT_BUILD_DIR=`pwd`/build/Ninja-ReleaseAssert/swift-
macosx-$(uname -m)
swift-source$ ninja -C ${SWIFT_BUILD_DIR} swift-benchmark-macosx-$(uname -m)
```

When modifying the testing infrastructure, you should verify that your changes pass all the tests:

```
swift-source$ ./llvm/utils/lit/lit.py -sv ${SWIFT_BUILD_DIR}/test-macosx-$(uname -
m)/benchmark
```