|  |
| --- |
| ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI |
| Yarn  A Speculative Multithreading System |
| Tests |
|  |
| **Rémi Attab** |
| **7/31/2011** |

|  |
| --- |
|  |

Table of Contents

[1. Introduction 3](#_Toc299922523)

[2. Functionality Testing 4](#_Toc299922524)

[2.1 Methodology 4](#_Toc299922525)

[2.2 Sequential Tests 4](#_Toc299922526)

[2.3 Parallel Tests 5](#_Toc299922527)

[3. Performance Tests 6](#_Toc299922528)

[3.1 Methodology 6](#_Toc299922529)

[3.2 yarnb Benchmark Description 6](#_Toc299922530)

[3.3 Benchmark Results 7](#_Toc299922531)

[A. Annexe 8](#_Toc299922532)

[A.1 Benchmark Result Table 9](#_Toc299922533)

[A.2 Benchmark Test Graph 10](#_Toc299922534)

# 1. Introduction

This document describes the testing infrastructure of the yarn project as well as presenting some of the tests results. Due to an unfortunate lack of development time, we were unable to construct a good testing framework for the yarn compiler tool. Because of this, this document will focus exclusively on the libyarn runtime component.

We will begin by describing the functionality tests that were constructed in order to ensure that our library behaves as expected. We will then examine the performance tests that were constructed in order to determine whether libyarn runtime component was fast enough to be used in a real world scenario.

# 2. Functionality Testing

In this section we will describe the functionality tests that were constructed in order to ensure that our library behaves as expected. We will begin with a discussion of the general testing methodology. We will then examine how the sequential and parallel tests where constructed.

## 2.1 Methodology

To build our tests we decided to use the C testing framework *Check* which covers all our required needs for the runtime component. Note that the compiler component would actually require a different set of tools provided by LLVM. The framework is also supported by our build environment and can be invoked using the *check* target.

Before any modifications to the code base can be committed it must first be buildable to ensure that we can checkout a patch at any point in the future without problems. Note that we allow for a module that is being build from scratch to be committed in an incomplete state. We allow this to ensure that we can periodically commit code that is in heavy development. It is still important that the source file not affect the overall build of the component and should therefore be removed from the build targets before being committed.

We also strongly recommend that all tests pass before committing any modifications. In the case where multiple issues are detected using the tests, it is still acceptable to commit each individual fixes separately in order to be able to easily revert any problematic fix.

Our general philosophy for writing test case is that we should focus more on integration tests than unit tests. The idea is that if a module A depends on a module B, then we should assume that module B works correctly when testing module A. If module B is deficient then we will detect it within the tests of module B. We do this because it would require too much effort to develop test harness for every module and we judge that it would add very little value.

## 2.2 Sequential Tests

Sequential tests were mainly put in place to ensure that the code is conformant to our expected behaviour. Every major module of the yarn component should have a test suite. Each test suite should contain at least one test case for every workflow. This also means that we don't test whether a single function behaves correctly, but we instead test whether every function calls in a workflow behaves as expected. By testing workflows, we end up testing each individual function in multiple ways while considerably reducing the number of test cases to write. This does have the side-effect of making it more difficult to identify a problematic area of code. Since all the difficult issues usually arise from the parallel test cases described in section 2.3, we think that it is more efficient to dedicate more time to the parallel tests then the sequential tests.

The following is a list of all the modules that have sequential test cases:

yarn\_map

yarn\_tpool

yarn\_pstore

yarn\_pmem

yarn\_bits

yarn\_tpool

yarn\_epoch

yarn\_dependency

## 2.3 Parallel Tests

Because the runtime component is meant to be executed in parallel, there are a considerable amount of requirements that can't be tested in purely sequential tests. It is therefore very important to have tests capable of detecting problems when the program is executed in parallel.

The parallel tests are also hosted on the *Check* framework but make very little use of its features. The framework is mainly used as a way to organize and execute the parallel test cases alongside the sequential test cases. Note that some issues can be difficult to reproduce and will in many instances only randomly manifest themselves. In order to reproduce these issues, we have created a small test script which is used to execute the parallel test cases in a loop until an error is detected. Experience has shown that it can take up to 300 repeated executions to reproduce a given issue. Because of this we highly recommend running this script for about a thousand cycles after having done any modifications that could affect the parallel behaviour.

The parallel tests all use the same format. They execute the task execution protocol described in the section 3.4 of the architecture document in order to calculate the sum of the numbers 1 to *n*. This test has the interesting property that we can easily check if the calculation were done correctly by checking the result with the mathematical formula:

The code being tested will usually also include many assert statements in order to test assumptions. If any of the *asserts* are triggered or if the final sum is not the expected sum, then we can determine that a problem occurred.

Parallel issues are usually very difficult to diagnose once it has been reproduced. Unfortunately, using a debugger is rarely useful because the issues are generally compounded over the execution protocol before the eventually show an abnormal behaviour. The only to work around this problem is to litter the code with print statements which we can then be used to find the problem. There is still a problem with that approach: most IO functions are by default thread-safe and will in many instances use locks to synchronize themselves. This can hide problems or make it much more difficult to reproduce problems. There are no known good solutions to this problem but it still recommended to always run the parallel tests **without** print statements before committing.

These are the modules that currently have parallel tests:

yarn

yarn\_epoch

yarn\_dependency

yarn\_tpool

# 3. Performance Tests

In this section we will describe the performance tests that were constructed in order to determine whether libyarn runtime component was fast enough to be used in a real world scenario. We will begin with a description of the methodology used to construct the performance tests. We will then describe the yarnb benchmarking program that was developed as part of this project and present the results obtained from this program.

## 3.1 Methodology

Our current approach to determine the performances of the runtime component is to determine for a given number of processors what workload is required per iteration in order to achieve a given speedup[[1]](#footnote-1) using a preset number of dependencies. A workload in our context is simply the time it takes to accomplish a task for a single iteration. If we compare the workload with a processor benchmark, we can determine how many cycles are needed within a single iteration in order to achieve a gain while using our speculative system. This scale needs to be applied to several use case scenario in order to get an accurate picture.

Note we should also eventually augment our benchmarking efforts by adding a series of real world programs that could be instrumented an executed. This could be used to confirm the results obtain from our other benchmark. Due to a lack of development time, this feature has been pushed back to a future release.

## 3.2 yarnb Benchmark Description

The yarnb program was developed to automatically calculate the workload test described in section 3.1. The benchmark program is capable of finding the workload required to achieve a given speedup using a given number of speculative threads. It is also able to calculate the maximum speedup reasonably achievable for a given number of speculative threads. Using these to capabilities it is capable of automatically calculating the workload required to achieve the speedups from a given minimum to the calculated maximum and for the number of speculative threads ranging from 1 to the number of core available on a machine.

yarnb emulates workload by simply spinning until it's workload time has been reached. yarnb is also capable of finding the workload required to achieve a speedup by doing an interpolation search[[2]](#footnote-2) over a range of workload times. Speedup calculation is accomplished using the following algorithm:

1. Time both the sequential task and parallel task *n* times each. Note that *n* is usually between 5 and 10.
2. Remove the greatest and smallest times and average the rest.
3. Divide the sequential average time by the parallel average time.

Note that this method could be improved but since, in most cases, the times remain fairly stable, we decided to keep the speedup calculation as simple as possible.

Note that currently yarnb is only capable of handling the ideal case where the number of rollbacks is kept at a minimum. We were unable to implement any more cases because we determined that yarnb would have to be completely rewritten in order to support this. This was deemed unacceptable because of time constraints. It is a planned feature for a future release of the yarn project.

## 3.3 Benchmark Results

The results of the benchmark execution are presented in annexe A1 and a graphical representation is presented in annexe A2. These results were obtained by executing the yarnb program on a 24 core server provided by the school.

The results demonstrate that we can't expect to achieve any speedup with a workload that is under 10μs. A quick test tells us that this corresponds to approximately 10 to 100 floating point divisions[[3]](#footnote-3). While this is a non-trivial workload, it does match the types of programs that we are targeting with our system. This test demonstrates that the yarn project is at least worth pursuing and developing further.

Another interesting phenomena that get be seen from the graph is that beyond 8 speculative threads, the speedup growth essentially stagnates at 5. We have tracked down the issue and it is caused by a bottleneck within the yarn\_epoch module. The bottleneck is caused by a lock that was added as a temporary fix to an issue that was raised by earlier benchmarking attempts. A solution is in active development and, once it is fixed, we expect the speedups to continue to grow with the number of processors. Unfortunately, we don't expect the solution to be ready by the end of this project.

# A. Annexe

## A.1 Benchmark Result Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Speedup | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 | 5 | 5.5 | 6 |
| 1 | 2811 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| 2 | 2225 | 7367 | 22324 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| 3 | 2210 | 5396 | 10159 | 18265 | 49006 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| 4 | 3140 | 6502 | 9110 | 13870 | 22420 | 41352 | 144829 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| 5 | 3986 | 8648 | 12466 | 16882 | 23173 | 30587 | 53256 | 138220 | 574862 | 1000000 | 1000000 | 1000000 |
| 6 | 5886 | 12703 | 16227 | 20409 | 28641 | 35111 | 46259 | 77159 | 174626 | 989920 | 1000000 | 1000000 |
| 7 | 8136 | 15240 | 20561 | 26884 | 33116 | 38216 | 46558 | 80388 | 194545 | 553284 | 1000000 | 1000000 |
| 8 | 9249 | 18673 | 26462 | 32905 | 39148 | 46567 | 60675 | 97545 | 142595 | 560985 | 998348 | 1000000 |
| 9 | 10395 | 21436 | 31045 | 37505 | 47188 | 54469 | 71211 | 100744 | 350605 | 931690 | 999987 | 1000000 |
| 10 | 13500 | 23484 | 32971 | 42244 | 53650 | 71091 | 88428 | 310867 | 591858 | 1000000 | 1000000 | 1000000 |
| 11 | 14600 | 25195 | 36285 | 45822 | 58274 | 72013 | 91663 | 291484 | 897308 | 1000000 | 1000000 | 1000000 |
| 12 | 13395 | 28238 | 38876 | 50182 | 61426 | 81452 | 132167 | 433504 | 977138 | 1000000 | 1000000 | 1000000 |
| 13 | 18886 | 32350 | 43072 | 56311 | 73479 | 93377 | 131288 | 385642 | 973286 | 1000000 | 1000000 | 1000000 |
| 14 | 20980 | 37529 | 49497 | 62239 | 81468 | 101899 | 140858 | 342437 | 1000000 | 1000000 | 1000000 | 1000000 |
| 15 | 23055 | 40937 | 55324 | 69686 | 84344 | 104659 | 134586 | 388291 | 1000000 | 1000000 | 1000000 | 1000000 |
| 16 | 24269 | 44104 | 59278 | 73520 | 89645 | 110575 | 150952 | 465645 | 1000000 | 1000000 | 1000000 | 1000000 |
| 17 | 25738 | 51321 | 64724 | 77869 | 97104 | 114640 | 141767 | 385893 | 1000000 | 1000000 | 1000000 | 1000000 |
| 18 | 27144 | 54078 | 71527 | 85637 | 99561 | 121458 | 155038 | 436422 | 986851 | 1000000 | 1000000 | 1000000 |
| 19 | 28297 | 58109 | 73255 | 88876 | 106446 | 126402 | 159788 | 415960 | 974226 | 1000000 | 1000000 | 1000000 |
| 20 | 34505 | 60668 | 79029 | 94576 | 111843 | 131009 | 152003 | 404860 | 939521 | 1000000 | 1000000 | 1000000 |
| 21 | 36044 | 62698 | 83620 | 100926 | 119332 | 138051 | 160165 | 433342 | 999811 | 1000000 | 1000000 | 1000000 |
| 22 | 38395 | 70484 | 89486 | 104037 | 123455 | 140625 | 166345 | 253737 | 879681 | 1000000 | 1000000 | 1000000 |
| 23 | 40222 | 69401 | 92305 | 112588 | 127884 | 146926 | 169627 | 419537 | 880698 | 1000000 | 1000000 | 1000000 |
| 24 | 38408 | 75154 | 98318 | 116255 | 133665 | 150741 | 171189 | 415551 | 991923 | 1000000 | 1000000 | 1000000 |

Note that the workloads are in nanoseconds and workloads over 100,000 microseconds are considered unreasonable and their associated speedups should be treated unachievable.

## A.2 Benchmark Test Graph

This graph is constructed from the table in annexe A.1.

1. A speed up represents how much faster the parallel program is compared to the sequential program. A speed up of 2 indicates that the parallel program is twice as fast as the sequential program. [↑](#footnote-ref-1)
2. A search algorithm based on linear interpolation (http://en.wikipedia.org/wiki/Interpolation\_search). [↑](#footnote-ref-2)
3. The widely inaccurate number is due to the difficulty of accurately measuring that number when fighting compiler optimizations. In order to get a more accurate number we would have to use a standardized benchmarking program. [↑](#footnote-ref-3)