Optimal Usage of the Computer Farm for Regression Testing

Daniel Hansson and Patrik Granath
Verifyter AB
Lund, Sweden
daniel.hansson@verifyter.com

***Abstract*— Companies spend more and more money on computer farms, yet there is never enough computer resources around to handle all ongoing projects. Some jobs are queued for a long time before they start executing and others crash due to lack of disk space, both of which affect project schedules.**

When management asks the engineers why there is a need to buy more computer farm resources there are no clear metrics, instead the answer is simply "we run more stuff". If additional computer resources are put in place they are consumed almost instantly, but no one reports that their projects can deliver earlier due to the extra computer resources available. It is very hard to do a cost-benefit analysis on computer farm resources.

This paper tries to solve this for regression testing, which is one of the largest usages of a computer farm. First we introduce some relevant metrics on *Cost* and *Quality*. Using these metrics we explore the optimal regression test setup. Finally we point to other areas where further optimization can be achieved.

The metric for computer farm *Cost* for a regression test setup is the total CPU test time. We want to find as many bugs as possible using as little computer farm resources as possible. The metric for *Quality* is the test fail ratio, i.e. how many tests fail vs the total number of tests run during a period of time.

We also introduce two secondary metrics, *Bug Identification Time*, i.e. how fast since the bug was introduced that we detect it. The more frequently we run the tests the faster we will detect bugs, but that has an adverse effect on Cost. After a bug has been detected it will be debugged and fixed. The total time from the bug was introduced until it was fixed is defined as the *Bug Fix Time*. These two metrics are considered secondary because the faster bugs are fixed the better for the Quality. They are not needed for the overall cost-benefit analysis, but they help us ensure that bugs are turned around fast, which is crucial to attain good Quality values.

Using these metrics we show that the optimal usage of the computer farm and for the project is to mix large regression test suites, which have good coverage, with short regression test suites, which have faster bug turnaround time and use less computer resources. This gives us lower Cost, shorter Bug Fix Time and higher average Quality.

Keywords—regression testing; computer farm

I. INTRODUCTION

A. *Usage of the Computer Farm Today*

During product development of ASIC's or software, new bugs are continuously introduced by mistake causing the quality of the product to deteriorate. These bugs are called regression bugs. Regression bugs are captured by running regular test runs, so called regression tests, which typically are RTL simulations that are run once or several times per day. These regression runs are typically run on a large computer farm, a resources that is shared with many other engineers within the company.

Today verification engineers try to run as much as possible on the farm to attain good coverage and fast detection of regression bugs, with very little feedback on what the cost of the computer farm is. This makes it also difficult to plan and discuss how much more computer farm resources are needed and the benefit of additional computer resources are not always clear as they tend to be consumed instantly when deployed.

Queued jobs on the farm, crashed jobs due to disk space limits and constant discussions, sometimes heated, about the computer farm is very common in today's project. The general view is that the more computer resources you have the easier it is for the projects to meet the release date.

There are no clear metrics that link the computer farm usage to the project schedules today, which means the discussions about extra resources are not always on firm scientific ground.

B. *Coverage vs Run-Time*

Verification engineers setup up the regression test suites with both coverage and computer resources in mind. The goal is to find bugs as fast as possible. However, the better a regression suite is at finding bugs, i.e. it has a good functional coverage, the longer time it takes to run, and this means there is a longer wait until a bug is identified. The choice of what exactly to run and at what frequency and what the cost is for the farm is today an area which is a lot based on gut feeling and educated guesses rather than a strict metric-based science.

Typically a short regression run can achieve say 50% relatively fast, but it takes much longer to reach higher levels of coverage. In figure 1 we show a typical example where a 2

hour regression test suite achieves about 50% coverage, but in order to achieve 80% coverage you need to run a 10 hour long regression test suite.

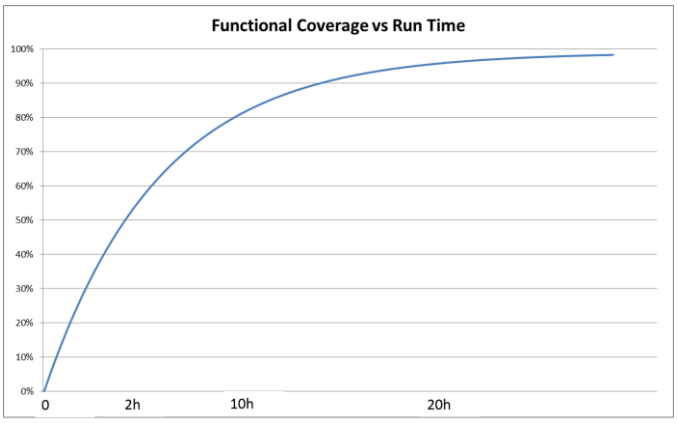

Figure 1. Example of Functional Coverage of a Regression Test Suite vs Run Time

The longer regression test suite can find more bugs due to higher coverage. In this example the longer regression test suite has 60% better coverage (80% / 50%), but it takes 400% longer time to run (10h / 2h) which means it is much slower at reporting bugs because a) it cannot be run as frequently as a short test suite and b) you have to wait for the complete test suite to finish in order to get the good coverage.

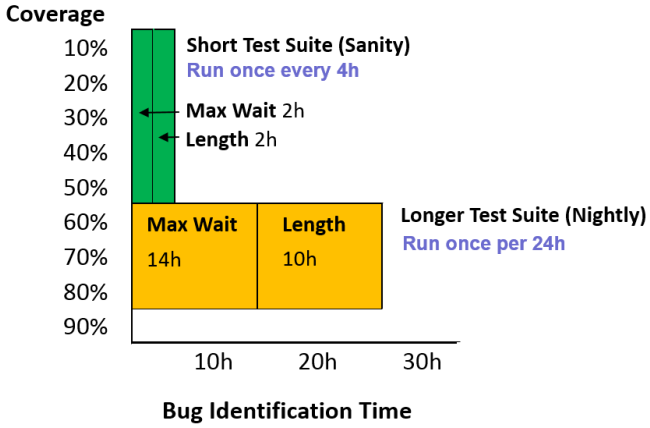

Figure 2. Bug Identification Times for short and long regression test suites

In Figure 2 we see the same two regression test suites and an example Bug Identification Time for usages which are typical in the ASIC industry. The short regression test suite (called Sanity) is 2 hours long and run every 4 hours. Best-case a bug is introduced just when the scheduled run is being kicked off, which means it will be reported after 2 hours, the time it takes to run the test suite. Worst-case it will take 4 hours before the bug is reported, if the bug is introduced two hours before the two hour run starts. The average Bug Identification Time is consequently 3 hours, i.e. the average between the best and worst cases.

The longer test suite which takes 10 hours to run and has 80% coverage is kicked off once per night and is consequently called a *Nightly* run. The best-case Bug Identification Time is 10 hours, worst-case it is 34 hours, thus the average Bug Identification Time is 22 hours.

This means, using the example in Figure 2 the longer nightly has a Bug Identification Time of 22 hours instead of 3 hours (633% slower), but will on average detect 60% more bugs because of higher coverage (80% vs 50%).

C. Launch Frequency vs Bug Identification Time

The more often you launch a test suite the less time there will be between a bug has been committed to the revision control system until a failing test is reported (this time is called the Bug Identification Time).

The shorter the test suite the more frequent it can be launched which in turn results in a shorter bug identification time. Shorter test suites can be launched more frequently and also you need to wait less time to get the full report.

Let's look at an example: a short test suite which takes only 2 hours to run and it is run once per 24 hour cycle. Assuming the bugs are committed in a uniform random way, this means the test suite will start on average 12 hours after the bug was committed. As the test suite takes 2 hours to run, the full report is available 14 hours after the bug was committed, i.e. the bug identification time is 14 hours. This is the left-most data sample in Figure 3.

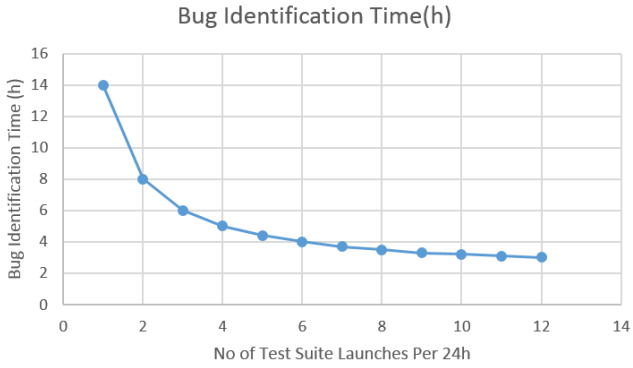

Figure 3. Bug Identification Time (hours) for different launch frequencies of a small test suite. The small test suite takes 2 hours to run and in this graph the test suite is launched 1 to 12 times per 24 hours cycle.

The cost of running the test suite is directly proportional to the launch frequency as each time you run the test suite it consumes the same number of CPU hours on the computer farm. Looking at Figure 3 you can see that see that it is very cost-efficient to run this small test suite say 1-5 times per day as the bug identification time is reduced fast with higher launch frequencies in this interval (14h for once per day down to 4.4h for 5 times per day). This is a big difference comparing to the

interval of 6-12 times per day where the difference in bug identification time is only 4h down to 3h. The more frequent you launch the test suite the shorter the bug identification time, but it is a case of diminishing returns.

D. Bug Fix Time

Bug Fix Time is here defined as the time from a bug is committed to the revision control system until a fix for this bug is committed at some time later (see Fig 4).

Bug Fix Time consists of two parts. The first part is the Bug Identification Time (BIT) which is the time from the bug is committed until a test failure is reported. During the Bug Identification Time a test suite is running, but not necessarily right after the bug has been committed as this depends on how often the test suite is launched.

The second part is here called the Debug Time, which is the time from a test failure is reported all the way until a fix has been committed. If debugging of regression bugs are done manually then there will not be a human being sitting and working with this issue for the complete length of the Debug Time. Typically it takes some time from the failure occurs until the right person has time to take a look at it but in this report we call the entire time from the test failure report until the fix has been committed for Debug Time.

The faster bugs are fixed the better. There are two ways of shortening the Bug Fix Time: 1) reducing the Bug Identification Time by selecting the optimal set of test suites and launch them at optimal intervals and 2) reducing the debug time by for example using an automatic debug tool[1,2].

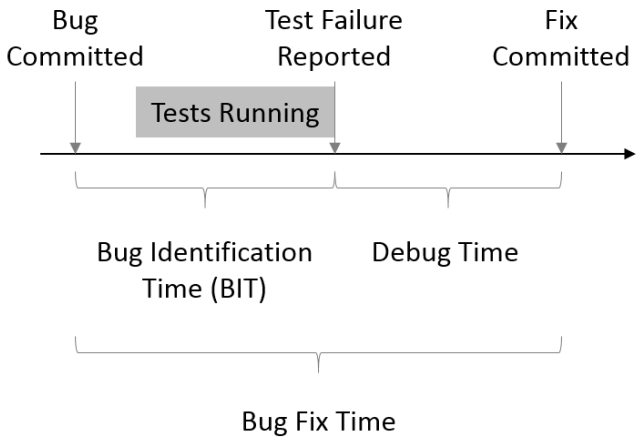

Figure 4. Bug Fix Time.

E. Test Fail Ratio

The faster the Bug Fix Time the lower the Test Fail Ratio. Having a low Test Fail Ratio means having a good quality of the device under test over time, which helps preventing the project from slipping, which is the whole purpose of regression testing. Ideally you want a Test Fail Ratio of 0, i.e. all tests are passing.

II. OPTIMISING THE USAGE OF THE COMPUTER FARM

A. Introducing Metrics

Let us define some useful metrics

Metric	Definition
Cost	Total CPU test time
Bug Identification Time	Time from a bug is committed to the revision control system until a failure is reported as a result of running a test suite
Bug Fix Time	Bug Identification Time + Additional debug time + until a fix has been committed to the revision control system
Test Fail Ratio (quality)	For a given period: The number of failing tests / total number of tests run. The lower the number the better the quality.

Table 1. Metrics for Regression Testing

B. Introducing Equations for Bug Identification Time and Cost when running one test suite

Let's start by defining the equations that calculates the bug identification time and the cost. The more frequently a test suite is run the shorter the bug identification time becomes at a higher cost.

If there is just one test suite then it is fairly straight-forward to calculate these metrics (see Fig 5). The average time from that a bug has been committed to the revision control system until the next launch of the test suite is called "Average Commit to Test Launch", abbreviated ACTL. If for example the cycle time we are looking at ("cycle") is 24h and there is just one launch ("L") per cycle then the ACTL is 12h. We are assuming that the bugs are inserted in a uniform random way during the cycle, which is why on average a test suite will be started 12h after a bug has been committed.

Bug Identification Time and Cost for 1 Test Suite

$$ACTL = \frac{cycle}{2 * L}$$

$$Average\ Test\ Suite\ Length\ (ATSL)$$

$$\boldsymbol{Bug\ Identification\ Time(BIT)} = ACTL + ATSL$$

$$\boldsymbol{Cost} = L * ATSL * Average\ CPU\ Usage\ (\#CPUs)$$

Figure 5. Equations to calculate Bug Identification Time (hours) and Cost (CPU hours) when running one single test suite

Continuing with the same example, still looking at the equations in Fig 5, the next step is to define the "Average Test

Suite Length" (ATSL). In this example let's say that the test suite length is 2h and because we only run one type of test suite the average test suite length also becomes 2h.

The Bug Identification Time (BIT) is simply the average commit to test launch (ACTL) which is 12h, added with the average test suite length (ATSL), which is 2h, thus making a total of 14h.

The Cost is the number of launches (L) multiplied with the average test suite length (ATSL) and the average parallel CPU's being used by the test suite. In this example let's say that 100 CPU's are used in parallel. Consequently the cost is 1 * 2h * 100 = 200 CPU hours.

C. Introducing Equations for Bug Identification Time and Cost when running two different test suites

Running two different test suites, with different test suite lengths and coverage, complicates the equations.

First of all, we make a simplification: we assume that the smaller test suite covers a subset of the functional coverage of the larger test suite. The only reason to run the smaller test suite is that it is faster and the only reason to run the larger test suite is that it has higher coverage. Also, we assume the functional coverage reflects the capacity to find bugs; halving the coverage means only half the number of bugs will be found.

What does this mean? First, let's call the short test suite "sanity" and the long test suite "nightly" and let's say the short test suite has 50% functional coverage and the nightly run has 70% functional coverage. In this case the sanity test suite is capable of finding 71.4% (50%/70%) of the bugs that the nightly run is capable of finding. If we run sanity more often and the nightly less often 71.4% of the bugs will be found earlier and the rest will be found later. In Fig 8 we refer to this value as the Δcoverage.

What we do with the equations in Fig 7 is to look at the different coverage tranches. Bugs found with the coverage of 50% can be found by both the sanity and the nightly run, whereas bugs in the 51%-70% coverage tranche can only be found by the nightly run.

An example of two test suites is shown in Figure 6. A short sanity run takes two hours to run and is run twice per day while a longer nightly run (8 hours) is run once per day. What is the average time from that a bug is committed to the revision control system until it is reported? Assuming a uniform bug distribution over time the bug can come at any point during the 24h cycle. The probability that a bug that is covered by both the sanity run and the nightly run (the so called *sanity coverage tranche*) will first be reported by the nightly test suite is 4/24 (max CTL_{StoN} / cycle) of the causes because there are only 4 hours per day where the next run is the nightly run (see the blue section marked "nightly" in the "Next launch" column). The chance on that other hand that the bug will be reported by a sanity run is 20/24, i.e. (max CTL_{StoS} + max CTL_{StoN})/cycle. This region is marked yellow in the "Next launch" column. The probability of being reported by either the sanity runs or the nightly runs depend on scheduling, i.e. at what time you have chosen to launch the respective test suite. That in turn depends on the lengths of the various test suites. The shorter the test suites the closer you can run them.

The column "Wait for Report" shows the waiting time between bugs being committed and reported, which is slightly different between the different types of test suites. The maximum waiting time or maximum Bug Identification Time, BIT (see max BIT_{StoS}, max BIT_{StoSN} and max BIT_{NtoS} in Fig 6) depends on scheduling but the minimum waiting time is determined completely by the length of the test suite. The latter happens when the bug was committed just before the checkout was done on which the test suite is sub-sequentially run.

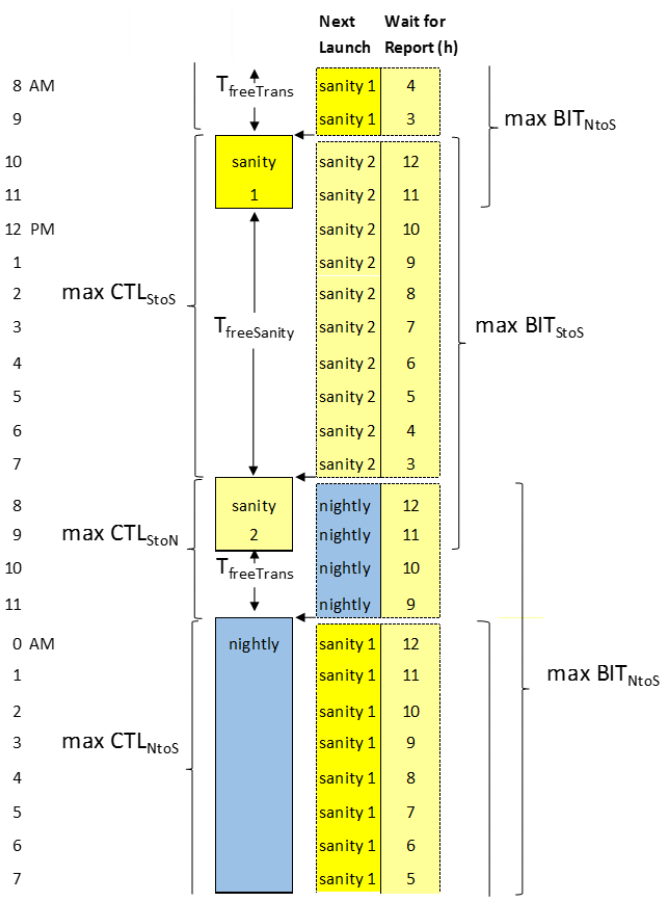

Figure 6. The max Commit To Launch (max CTL) and max Bug Identification Time (max BIT) for the 3 sequences of test suites (sanity to sanity, sanity to nightly, nightly to sanity) when having 2 Test Suites ("sanity", "nightly").

Calculating total Bug Identification Time (BIT) for both the test suites involves two steps: 1) to calculate the BIT for the sanity coverage tranche and the BIT for the nightly coverage tranche separately and 2) adding them together according to their weight given by the coverage.

Calculating the BIT for the nightly coverage tranche is easy because there is only one test suite, the nightly test suite,

which covers this tranche. Consequently we can use the same equation as we did for one test suite (compare Fig 5 with the definition of $BIT_{nightly}$ in Fig 7).

Calculating the BIT for the sanity coverage tranche is more elaborate as we have to take into account two test suites, sanity and nightly, which both covers the sanity coverage tranche. Having two different test suites means there are 3 possible sequences between them: sanity followed by sanity, sanity followed by nightly and nightly followed by sanity. This is true for any number of sanity and nightly test suite runs as long as 1) there are fewer nightly runs than sanity runs and 2) the test suites are interleaved as much as possible. Fulfilling these two conditions makes sense because both produce a lower overall bug identification time. Note that if both conditions are fulfilled then there is no sequence where a nightly run is followed by another nightly run.

For each of the 3 test suite sequences we need to 1) calculate the probability that a bug appears in one of these 3 test suite sequences multiplied with 2) the average BIT for each of the 3 sequences. This is reflected in the equation for BIT_{sanity} in Fig 7 where each line represents one of the 2 test suite sequences, starting with sanity to sanity. For both the probability and the average BIT the key value is the max commit to launch, max CTL, which is the time from a bug is committed until the next test suite is launched. The max CTL depends on the length of the previously launched test suite plus the free time when no test suites are launched until the next test suite is launched. The max CTL for the 3 test suite sequences are shown in both Fig 6 and Fig 7.

To calculate the probability you take max CTL for one of the 3 test suite sequences multiplied with how many times this sequence occurs per cycle divided by the cycle time. This gives you the proportion of the cycle time that you are in one of the 3 test suite sequences (see BIT_{sanity} in Fig 7). Note that $L_{nightly}$ reflects how many times you transition from sanity to nightly (or the other way around) because one of the conditions for the equations is that the test suites are interleaved. Consequently (L_{sanity} - $L_{nightly}$) reflects how many times the test sequence from sanity to sanity occurs, because those are the only once that are not transitions to or from the nightly run.

The other part of the formula for the BIT_{sanity} in Fig 7 is the average BIT for each of the 3 test sequences. The average BIT is the average commit to launch (max CTL / 2) plus the test suite length (TSL) of the subsequent test suite.

The last step is to calculate the Bug Identification Time (BIT) where the BIT_{sanity} and $BIT_{nightly}$ weighed together according to their contribution to the two different coverage tranches called "sanity" and "nightly". Also the cost is calculated by adding the cost of each test suite separately.

Max Commit to Launch for the 3 different sequences of test suites (sanity –> sanity, sanity -> nightly, nightly -> sanity):

$$max\ CTL_{StoS} = TSL_{sanity} + T_{freeSanity}$$
$$max\ CTL_{StoN} = TSL_{sanity} + T_{freeTrans}$$
$$max\ CTL_{NtoS} = TSL_{nightly} + T_{freeTrans}$$

Bug Identification Time (BIT) for the sanity coverage tranche:

Average BIT for each of the 3 sequences ↓ Probability for a bug to appear in this part of the cycle ↓

$$BIT_{sanity} = \left(\frac{max\ CTL_{StoS}}{2} + TSL_{sanity}\right) * \left(\frac{max\ CTL_{StoS} * (L_{sanity} - L_{nightly})}{cycle}\right) +$$
$$\left(\frac{max\ CTL_{StoN}}{2} + TSL_{nightly}\right) * \left(\frac{max\ CTL_{StoN} * L_{nightly}}{cycle}\right) +$$
$$\left(\frac{max\ CTL_{NtoS}}{2} + TSL_{sanity}\right) * \left(\frac{max\ CTL_{NtoS} * L_{nightly}}{cycle}\right) +$$

Bug Identification Time for the nightly coverage tranche:

$$BIT_{nightly} = \frac{cycle}{2 * L_{nightly}} + TSL_{nightly}$$ (one test suite equation)

Bug Identification Time Total:

$$\begin{cases} \Delta coverage_{sanity} = \dfrac{coverage_{sanity}}{coverage_{nightly}} \\ \Delta coverage_{nightly} = \dfrac{coverage_{nightly} - coverage_{sanity}}{coverage_{nightly}} \end{cases}$$

$$\boldsymbol{Bug\ Identification\ Time_{total}} = (\Delta coverage_{sanity} * BIT_{sanity}) + (\Delta coverage_{nightly} * BIT_{nightly})$$

$$\begin{cases} Cost_{sanity} = L_{sanity} * TSL_{sanity} * \#CPUs_{sanity} \\ Cost_{nightly} = L_{nightly} * TSL_{nightly} * \#CPUs_{nightly} \end{cases}$$

$$\boldsymbol{Cost_{total}} = Cost_{sanity} + Cost_{nightly}$$

Figure 7. Equations to calculate the Bug Identification Time (BIT) and Cost for 2 Test Suites ("sanity", "nightly")

What is the optimal scheduling for the two test suites? The max CTL matters both for the probability of being in one of the 3 test sequences and also for the average BIT for that test sequence. There is consequently a square dependency on the max CTL for each test sequence, which means there is an optimal solution to be found for the BIT_{sanity}.

Using the Lagrange multiplier [3] we find the optimal solution for the BIT_{sanity} (see Fig 8). The constraint we are using is that the sum of the max CTL for the 3 test sequences equals the cycle, which is something that you can see an example of in Fig 6. The first step using the Lagrange multiplier is to calculate the partial derivatives for BIT_{sanity} as well as the constraint formula G (see Fig 8). The second step is to plug these derivatives into the Lagrangian formula (gradient BIT_{sanity} = λ * gradient G). The result is that the optimum value is achieved when (max CTL + TSL) for each of the 3 test suite sequences equal each other. Another way to

express this is that the max BIT for each test sequence should equal each other (see Fig 6 where both the max BIT and max CTL for each test suite sequence is marked). Solving these equations (step 3 in Fig 8) we are able to define the optimal free times between the test suites, when nothing should run: $T_{freeSanity}$, the free time between sanity runs and $T_{freeTrans}$, the optimal free time between nightly and sanity runs.

In Fig 8 you can see that $T_{freeSanity}$ is set to the total available free time plus a compensation factor divided by the total number of runs. Correspondingly $T_{freeTrans}$ is set to the total available free time minus a compensation factor divided by the total number of runs. This compensation factor is set to the difference in test suite lengths multiplied with the number of test suite launches that is not related to the test suite sequence whose free time is being calculated.

Optimal placement for 2 Test Suites ("sanity", "nightly")

We assume optimal ordering and optimal launch frequencies, i.e. :

- $L_{nightly} <= L_{sanity}$
- Sanity and nightly runs are interleaved as much as possible

Optimising BIT_{sanity} using a Lagrangian multiplier:

constraint: $G = max\ CTL_{StoS} + max\ CTL_{StoN} + max\ CTL_{NtoS} = cycle$

Step 1. Find partial derivatives for BIT_{sanity} and the constraint (G)

$$\frac{d(BIT_{sanity})}{d(CTL_{StoS})} = \frac{maxCTL_{StoS} + TSL_{sanity}}{cycle} \qquad \frac{d(G)}{d(maxCTL_{StoS})} = 1$$

$$\frac{d(BIT_{sanity})}{d(CTL_{StoN})} = \frac{maxCTL_{StoN} + TSL_{sanity}}{cycle} \qquad \frac{d(G)}{d(maxCTL_{StoN})} = 1$$

$$\frac{d(BIT_{sanity})}{d(CTL_{NtoS})} = \frac{maxCTL_{NtoS} + TSL_{sanity}}{cycle} \qquad \frac{d(G)}{d(maxCTL_{NtoS})} = 1$$

Step 2. Plug into the Lagrangian formula (gradiant BIT_{sanity} = λ * gradiant G)

$$\frac{maxCTL_{StoS} + TSL_{sanity}}{cycle} = \lambda$$

$$\frac{maxCTL_{StoN} + TSL_{nightly}}{cycle} = \lambda$$

$$\frac{maxCTL_{NtoS} + TSL_{sanity}}{cycle} = \lambda$$

Each (max CTL + TSL) should be equal to each other to get min BIT_{sanity}

Step 3. Solving the equations to get optimal free time between runs

Total free time when no test suites are running

$$T_{freeTot} = cycle - (TSL_{sanity} * L_{sanity}) - (TSL_{nightly} * L_{nightly})$$

$$L_{tot} = L_{sanity} + L_{nightly}$$

$$L_{sanToSan} = L_{sanity} - L_{nightly}$$

Optimal free time between two sanity runs

$$T_{freeSanity} = \frac{T_{freeTot} + (L_{tot} - L_{sanToSan}) * (TSL_{nightly} - TSL_{sanity})}{L_{tot}}$$

Optimal free time between one sanity run and one nightly run

$$T_{freeTrans} = \frac{T_{freeTot} - L_{sanToSan} * (TSL_{nightly} - TSL_{sanity})}{L_{tot}}$$

Figure 8. Equations to calculate the optimal placement for 2 Test Suites ("sanity", "nightly"). The result is the optimal free time when nothing is being run between two sanity runs ($T_{freeSanity}$) and the optimal free time between a sanity run and a nightly run ($T_{freeTrans}$). The Lagrange multiplier is used to prove that the min BITsanity is achieved when (max CTL + TSL) for each of the 3 test sequences are equal to each other.

D. Introducing Equations for Bug Identification Time and Cost when running three different test suites

Running three different test suites expands on the equations described when running two test suites, but the principle is the same. Now there are three test suites, with different test suite lengths and coverage, and consequently there will be 3 coverage tranches which the equation needs to look at.

Another difference is that there are now two types of total free time between the test suite runs: one which takes into account the test suites that affect the sanity coverage tranche (all test suites) and the other which takes only account of the test suites affecting the nightly tranche (nightly and weekend).

A third point which needs expanding is the number of transitions, which was 2 in the case of 2 test suites and are now 9 in the case of 3 test suites.

We are not presenting the formulas in this section as they becomes very large, but there are no new principles as such. We have implemented them in computer program and the results for the 3 test suites is presented later in this paper.

E. Bug Fix Time

The Bug Fix Time is simply the sum of the Bug Identification Time plus the Debug Time.

F. Test Fail Ratio

The Test Fail Ratio is defined as all failing tests for a given period, divided by all tests that was run in this period. See Fig 9. For example, if 10 tests are run each day and 1 test fails every other day then the Test Fail Ratio would be 5%.

$$\text{Test Fail Ratio} = \frac{\text{failing tests}}{\text{all tests}}$$

for a given time period

Figure 9. The Test Fail ratio is failing tests divided by all tests for a given period.

III. METHODOLOGY

A. How the measurements was done

We implemented the equations in an excel sheet where we calculated exhaustively all combinations. We sorted the results after cost and then after bug identification time. This allowed us to create a sub-list of the optimal regression test setup for each cost. We present this both as a list and as a graph.

IV. RESULTS

The result is a model (based on the equations) which allows us to optimize each specific regression test setup. By entering the

test suite lengths, the parallel CPU usage and the coverage for the test suite(s) the model will provide the optimal launch frequency for each test suite.

A. *One Test Suite*

When there is only one test suite used for regression testing then the only question is how frequently it should be launched compared to what it costs. The cost is directly proportional to the number of launches. It does not matter what the coverage is or what the average parallel CPU usage is when there is only one test suite in order to answer this question. Also, the test suite time only matters to determine the max number of launches per cycle, but it tells you nothing about the optimal usage.

As the bug identification time only depends on how often it is launched the graphs looks very similar for different scenarios (see Fig 10 and Fig 11). Consequently we can formulate a generic answer when you are running only one test suite:

- Up to about 5 times/cycle: The Bug Identification Time is substantially reduced for each extra time you run the test suite.
- From about 6 times/cycle: The Bug Identification is still reduced for each extra time you run the test suite, but only by a fraction of the test suite length

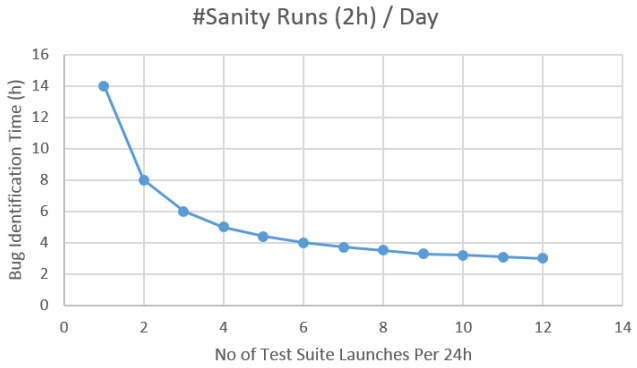

Figure 10. Sanity Run: One short test suite (2h) that is launched X times per day

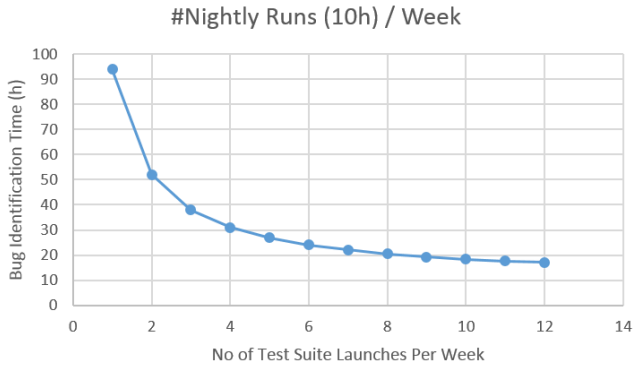

Figure 11. Nightly Run: One longer test suite (10h) that is launched X times per week

B. *Two Test Suites*

When there are two test suites, with different test suite lengths and different functional coverage then it becomes more complicated (see Fig 12).

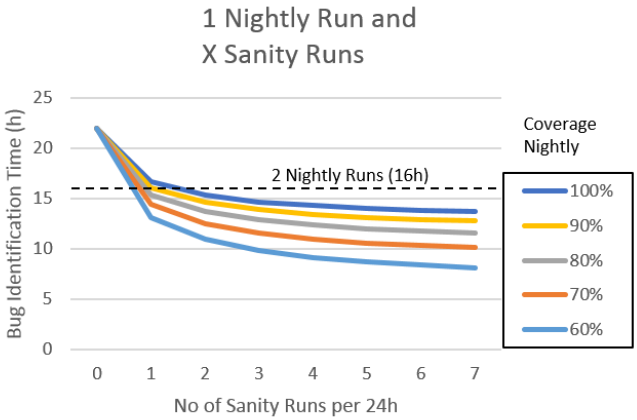

Figure 12. Two test suites: a nightly run (10h) and a sanity run (2h, 50% coverage). The graph shows the bug identification time depending on how often the sanity is run for each time the nightly is run. It also shows the effect of differences in functional coverage, the coverage for the nightly run varies between 60% and 100% while the coverage for the sanity run is fixed at 50%.

Fig 12 shows how the bug identification time depends on 1) how often you run the test suites and 2) the difference in functional coverage between the larger "nightly" test suite and the smaller "sanity" test suite.

The graph shows that the more often you run the "sanity" test suite (the x-axis), the lower the bug identification time is. In all data points the nightly test suite is run once per 24h and the sanity is run as many times as indicated by the x-axis. E.g. at 0 sanity runs there is only one nightly run per 24h and no sanity runs.

The graph shows also shows the impact of the difference in coverage between the sanity and the nightly test suites. The sanity run is fixed at 50% functional coverage and the coverage of the nightly run varies between 60% and 100%. The only thing that matters is the relation between the test suites. When the nightly test suite has 100% coverage then the assumption is that it will reveal twice as many bugs as the sanity test run which has 50% coverage. The graph shows that the larger the difference is in terms of coverage between the test suites the worse it is, i.e. the bug identification time becomes longer. The reason is that a higher portion of the bugs will only be discovered by the nightly run which is run less frequently.

What conclusions can we make from the graph? The first conclusion is that it is always better to run the longer test suite (nightly) just once and the shorter test suites (sanity) several times per cycle. In Fig 12 you can see that it is only better to run the nightly test suite twice (the dashed line) – and the sanity 0 times - when the difference in coverage is close to double (90%-100%) that of the sanity run (50%) and when the sanity run is only run once. However, if you run the sanity test

suites twice or more per cycle then this always provides a better result.

The second question is how often you should run the sanity run in such a setup. The coverage differences between the test suites affects the answer but not in any way that fundamentally changes the question. You should aim to run the sanity run roughly 2-4 times per cycle and the nightly run once. In this range it is beneficial for each extra run you do, but after that it quickly drops off. Compare this to the conclusion for one test suite which was to run "up to about 5 times/cycle". Running the sanity test suite 4 times plus the nightly once means 5 test suite runs per cycle.

Note that these conclusions did not consider the cost of running the two test suites, which is much larger for the nightly test suite than for the shorter sanity test suite as it uses less CPU hours and licenses. The cost makes no difference in this case as we reached the conclusion to just run the more expensive nightly test suite once by just looking at the bug identification time. We must run the nightly run at least once, otherwise there is a loss of coverage.

C. Three Test Suites

Having three test suites of different lengths and coverage means there are many different combinations of setting up the regression runs. The model is however able to list the best alternatives in order of cost.

In this example there are three test suites: sanity (length 2h, 40% coverage), nightly (length 10h, 60% coverage) and weekend (length 40h, 70% coverage) and the cycle is 1 week.

Cost	BIT	Launch Frequencies		
(x-axis)	(y-axis)	Sanity	Nightly	Weekend
52	66.26	1	1	1
54	56.05	2	1	1
56	52.88	3	1	1
58	50.76	4	1	1
60	49.25	5	1	1
62	48.12	6	1	1
64	47.24	7	1	1
66	45.91	3	2	1
68	44.38	4	2	1
70	43.24	5	2	1
72	42.35	6	2	1
74	41.64	7	2	1
76	41.06	8	2	1
78	40.58	9	2	1
80	39.92	5	3	1
82	39.21	6	3	1
84	38.63	7	3	1
86	38.15	8	3	1
88	37.74	9	3	1
90	37.39	10	3	1
92	37.09	11	3	1
94	36.68	7	4	1
96	36.27	8	4	1
98	35.93	9	4	1
100	35.63	10	4	1
102	35.36	11	4	1
104	35.13	12	4	1
106	34.92	8	5	1
108	34.63	9	5	1
110	34.37	10	5	1
112	34.14	11	5	1
114	33.94	12	5	1
116	33.76	13	5	1
118	33.60	14	5	1
120	33.41	10	6	1
122	33.21	11	6	1
124	33.04	12	6	1
126	32.88	13	6	1
128	32.74	9	3	2
130	32.48	10	3	2
132	32.10	6	4	2
134	31.78	7	4	2
136	31.50	8	4	2
138	31.27	9	4	2
140	31.06	10	4	2
142	30.87	11	4	2
144	30.60	7	5	2
146	30.30	8	5	2
148	30.12	9	5	2
150	29.95	10	5	2
152	29.80	11	5	2
154	29.67	12	5	2
156	29.33	8	6	2
158	29.18	9	6	2
160	29.05	10	6	2
162	28.93	11	6	2
164	28.82	12	6	2
166	28.73	13	6	2
168	28.60	9	7	2

Figure 13. The optimal launch frequency for three test suites in order of cost. For example, if you want a Bug Identification Time of max 33h then the best setup is to run the Sanity run 13 times per week, the Nightly 6 times per week and the Weekend run just once per week. This setup will run some test suite 75% of the time (126h out of 168h, which is a full week).

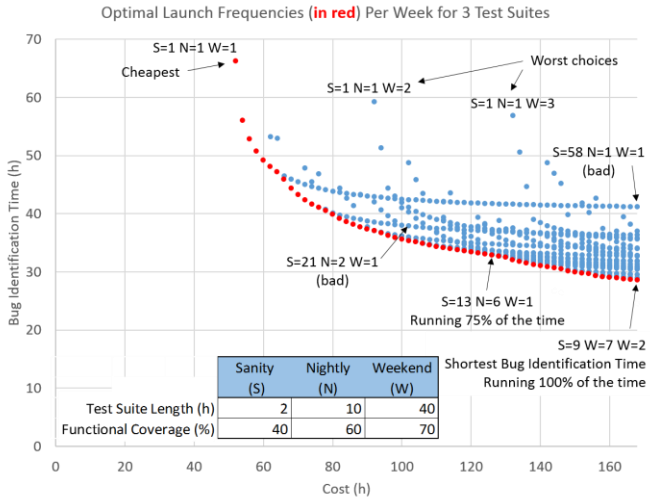

Figure 14. Optimal launch frequencies for three test suites in order of cost. This graph uses the same values as the table in Fig 10.

In Fig 14 you can see that the benefit of higher costs starts to flatten out in a similar pattern as we saw for 1 and 2 test suites. However now we have many data series on top of each other that together covers all possible combinations of launch frequencies. You can distinguish the individual data series in Fig 14 starting at S=1 N=1 W=1 and ending at S=58 N=1 W=1, which follows the same patterns we have seen earlier in this paper. Only first combinations of this data series is optimal (marked red), before it is overtaken by the next data series which has N=2 and W=1 (one of its data points, S=21 N=2 W=1 is highlighted).

If you look in the table in Fig 13 you will see that all optimal combinations of launch frequencies follows this familiar pattern. There is a lot to gain to pick a combinations of launch frequencies with a bug identification time lower than roughly 40 hours as it is very cheap to achieve major improvements in the bug identification time for the lower cost options. In this region the number of nightly runs is 1.5 – 6 times higher than the number of weekend runs and the number of sanity runs is 1.3 – 3.7 times higher than the number of nightly runs.

Note that both Fig 13 and Fig 14 only allow up to 168 hours of run as this is the number of hours per week and no overlap is allowed between the regression test suites. Having test suites of different sizes running at the same time is less optimal than making sure they do not overlap in time. If two test suites do not overlap in time then the larger test suite will cover the same area as the shorter test suite (and more), which for those bugs that are revealed by this overlapping coverage would be like running the smaller test suite one extra time. If overlap is allowed this benefit is lost.

Another comment is that in this example we assume that all test suites are using the same number of CPU's in parallel. The only difference are the lengths of the test suites. This is not a limit of the equations presented earlier, it is just an assumption in this example.

What data point should we select in Fig 14? Any choice on the red line is an optimal choice which is th-+

e best choice for a given cost. How many test hours (cost) are you willing to spend? The answer to that is probably somewhere between running 50% of the time (84h) because you get so much value for money up to roughly that time and less than 100% of the time. Even if you want to run 100% of the time it may not be possible because of the load on the computer farm. There is always some amount of queueing during peak hours. In this example let's say we want to run some kind of test suite 75% of the time, in which case the optimal choice is S=13 N=6 W=1.

Using this example we get a feeling for how to pick an optimal combination of launch frequencies. However to get the exact optimal setup for a specific case you need to feed in the data into the model.

D. Test Fail Ratio vs Bug Fix Time

The faster bugs are fixed the better for the quality. The Test Fail Ratio (i.e. the quality over time) is directly proportional to the Bug Fix Time (see Fig 15).

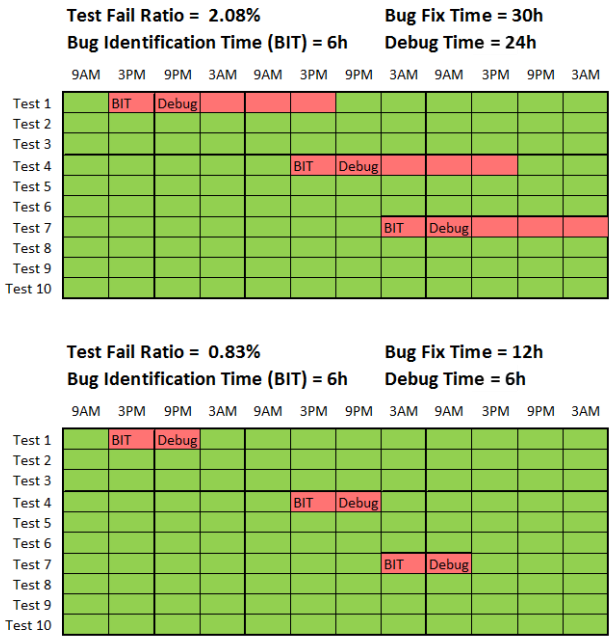

Figure 15 The Test Fail Ratio (i.e. the quality over time) is directly proportional to the Bug Fix Time. In this example the Bug Identification Time is the same in both scenarios but the debug time is 4x shorter in the lower scenario, which leads to a 2.5x shorter Bug Fix Time, which in turn leads to a 2.5x lower Test Fail Ratio.

This The Bug Fix Time can be shortened by shortening either the Bug Identification Time or the Debug Time. Figure 15 shows an example of how by only shortening the Debug Time this greatly affects the Test Fail Ratio.

This is important as the Test Fail Ratio affects the length of the project. The fewer issues there are the faster the project can advance as there are less hurdles to overcome. Ultimately the goal is to release the product when the Test Fail Ratio is at a minimum, preferably at 0. The whole reason that regression testing is being performed is to keep the Test Fail Ratio low, i.e. to quickly detected and fix issues, in order for a project to release as early as possible.

This paper has shown that the way to do this is to optimize the Bug Identification Time vs the Cost and by reducing the Debug Time.

REFERENCES

[1] PinDown from Verifyter, http://www.verifyter.com

[2] Onpoint from Venssa, http://www.venssa.com

[3] Langrange Multiplier: https://en.wikipedia.org/wiki/Lagrange_multiplier