ISSN:0975-9646
Siva Kumar Kotamraju et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 7 (6) , 2016,2469-2471

Optimization of Constraint Engine

Siva Kumar Kotamraju

CSE Department,
Vignans Nirula Institute of Technology & Science for Women
Pedapalakaluru, Guntur
Andhra Pradesh, India-522005

***Abstract*— The Performance Analysis of Constraint Engine is the work done to discover new Optimization methods for Optimizing Perl modules**

***Keywords*— Optimization, Constraint Engine**

I. INTRODUCTION

In order to determine if tests can be merged together into a single job it is necessary to know the requirements of the template to run correctly, and resolve the competing requirements of templates for shared resources such as memory locations, template build options, API options and DUT configuration settings. The constraint engine provides the functionality of taking in all the requirements for tests and finding a solution that meets the user's requests both in terms of absolute requirements and also in terms of probabilistic goals set by the users. Constraint Engine is part of Migi Tool

II. GOALS

1. Track usage of shared resources to avoid building tests that cannot run together successfully

This is the minimum functionality – without this we can't do test merging without the majority of merged tests failing due to conflicts over shared resources such as memory, shared control registers and APIs that might affect or check more than one thread.

2. Provide globally optimal allocation of shared resources where possible

We want to make globally-aware decisions about the shared resources (and decisions that affect a test's use of those shared resources) so that a random choice doesn't unnecessarily restrict our ability to run tests. Historical experience from MTMerge and previous MP test generation tools strongly indicate that non-global allocation in this space is undesirable.

3. Provide tests dynamically

Determining the runtime of a test is impossible without running the test on the MUT (it's the halting problem.) We also believe that it is extremely likely that we will often have threads on the MUT that will be idle for nontrivial periods of time. We wish to reuse that simulation time to do useful testing, so the constraint engine must be capable of allocating resources to a new test after other tests have been deployed on the system.

4. Provide statistically significant solutions

Users will be specifying parameters that can take on one of many values, and in many cases the distribution of those values over many runs of tests is significant to the user. While in some cases it is impossible to provide the distribution to multiple values, there needs to be a 'best case' effort by the tool to provide the distribution requested.

5. Provide an interface for selecting random values with constraints in a test

The constraint engine also will provide a mechanism for tests to select random values for build parameters and specify constraints on those parameters within a single test. This is a useful mechanism for test authors to more easily add randomness to their tests while putting limits on the randomness.

III. GROUNDWORK AND DEFINITIONS

Parameters and Options

A parameter is a variable that is assigned a value by the CE. An option is a possible value that the parameter may take. Parameters may have multiple options, and the options can have weights which affect the probability of which option is chosen.

An option can depend on the value of other parameters. All options will be evaluated by the Perl interpreter and therefore may contain arbitrary Perl code that references the values of other parameters.

Constraints

A constraint is a restriction that is imposed on the space of parameter values. Constraints are in the form of arbitrary Perl code that will be evaluated. If the code evaluates to true (non-zero), the constraint is said to be satisfied.

Streams

A stream is a series of software instructions that executes on a single LP (logical processor). It will contain parameters, and constraints that pick combinations of parameter values the stream needs to run.

Tests

A test is a container for one or more streams. Tests can also contain parameters and constraints that pick combinations of parameter values the test needs to start building.

Tests waiting to run are kept in a priority queue. This means that in general, higher priority tests will run before lower priority tests as long as constraints are not blocking the higher priority tests.

Test blocks

A test block is a container for one or more tests. Test blocks can be layered, meaning containing other test blocks. Constraints can also be specified for test blocks, with the intention that those constraints will be copied into every stream inside, and evaluated inside the stream.

Merge blocks

A merge block is a container for one or more tests. There are two types:

Parallel merge blocks contain tests that should run concurrently. For any test in the merge block to build and run, all tests in the merge block must also build and run.

Serial merge blocks contain tests that should run in series, one after another, in the order specified.

Merge blocks can also contain other merge blocks, meaning merge blocks can be layered.

Implementing serial merge blocks is difficult, and may not be done unless there is demand for the feature. Furthermore, from here, if the term merge block is used without specifying the type, it is referring to parallel merge blocks

Test Queues

Every test has a status, which can be one of the following:

RUNNING The test has been reported as running by the Migi Control. For a test to reach this state, it must have built correctly, meaning all of its constraints are satisfied and all of its parameters are "locked" (they can no longer change their value.)

BUILDING The test is either ready to build or currently building. For a test to stay in this queue, all of its constraints must be satisfied. Updates to parameters throughout the build process may cause constraints to fail, which would cause removal of the test from the BUILDING queue.

WAITING The test is currently failing some of its constraints and cannot be built right now. The failing constraints could be system-imposed (no logical processors are available for the test), or user-imposed (the user may have requested the test run alone on a core.)

WAITING -> BUILDING (promotion)

When logical processors in the system are available, some WAITING tests will be promoted to BUILDING. The CE will then attempt to solve the constraint system. If successful, the tests added to the BUILDING queue will remain, indicating they are ready to build. If adding the new tests causes the constraint system to fail, tests will be removed (and put back in the WAITING queue) until the constraint system can be solved.

BUILDING -> RUNNING (promotion)

When the Migi Control indicates a test will run, it is promoted to RUNNING. Once a test is in the RUNNING queue, it will remain there until the Migi Control tells the CE to release the test (probably because the test finished running.)

BUILDING -> WAITING

This transition can happen in two cases

1) If the Migi Builder updates a test's parameters, this may cause the constraint system to fail. In this case, the test goes back to WAITING.

2) As logical processors become available, some WAITING tests will be promoted to BUILDING. If the CE cannot solve the constraint system with these new tests, the tests will return to WAITING. Tests are returned to the WAITING queue lowest priority first, until the constraint system can be solved.

Parameter / Constraint Interdependencies

Parameters and constraints are concepts with many interdependencies. The following relationships exist:

1) Constraints are arbitrary sections of Perl code that check whether parameters satisfy certain properties. As such, they are dependent on the parameters they reference.

The CE must be aware of the parameters a constraint is dependent on so that if a constraint evaluates to false (indicating that it is unhappy with one or more of the parameters), the CE can then pick different options for some of the dependent parameters in an effort to satisfy the constraint.

2) Parameters are related to constraints. If a parameter changes, the CE must know all of the constraints that referenced that parameter so that those constraints can be re-evaluated.

3) A parameter can be dependent on other parameters. Because an option for a parameter may depend on the values of other parameters, changing one parameter can affect the values of several other parameters. Every time the CE changes the value of a parameter, it needs to check which other parameters depend on it, so those parameters can be re-evaluated.

IV. OPTIMIZATION TECHNIQUES:

1) String Comparison:

The code inside the subroutine does arithmetic and a numeric comparison of two strings. It assigns one string to another if the condition tests true but the condition always tests false.

```
     sub test_code{
  my ($a,$b) = qw(foo bar);
  my $c;
  if ($a == $b) {
    $c = $a;
  }
 }
```

Now let's fix the comparison the way it should be, by replacing == with eq, so we get:

```
my ($a,$b) = qw(foo bar);
  my $c;
  if ($a eq $b) {
    $c = $a;
```

2) Concatenation:

$title = 'My Web Page';

print '<h1>' . $title . '</h1>'; # Concatenation (slow) .

print '<h1>', $title, '</h1>'; # List (fast for long strings)

3) Way of Calling Subroutines:

- Light Subroutines: function form (like Foo::bar('Foo'))
- Heavy Subroutines: method form (like Foo->bar())

It will make code easier to develop, maintain and debug, saving programmer time which, over the life of a project may turn out to be the most significant cost factor

4) Needless importing :

use POSIX; # Exports all the defaults -0.516us

use POSIX (); # Exports nothing -0.315us.

5) Avoid $&:

$text =~ /.* rules/; $line = $&; # Now every match will copy $& slow

$text =~ /(.* rules)/; $line = $1; # Didn't mention $& fast

6) Pull things out of loops:

Perl's hash lookups are fast. But they aren't as fast as a lexical variable. The value of $type didn't change, so I pulled the lookup out above the loop into a lexical variable.

my $type_func = $encode_types{$type};

7) Transliteration operator:

- tr/!// # fastest way to count chars: In scalar context it returns the number of characters that matched. It's the fastest way to count the number of occurrences of single characters and character ranges
- tr/q/Q/ faster than s/q/Q/g: tr is also faster than the regexp engine for doing character-for-character substitutions.
- tr/a-z//d faster than s/[a-z]//g: tr is faster than the regexp engines for doing character range deletions

8) String Concatenation:

- Ordinary Concatenation: less time
- Generating an Array and concatenating with join: More time

9) Use references : While working with large arrays or hashes and use them as arguments to functions. Saves memory

10) String handling: In a Web application, use single quotes rather than doubles.

11) Loops: Excessive function calls in a loop are generally a bad idea. Loop in a Function is a good Procedure. Use map instead of for-each for each Pass.

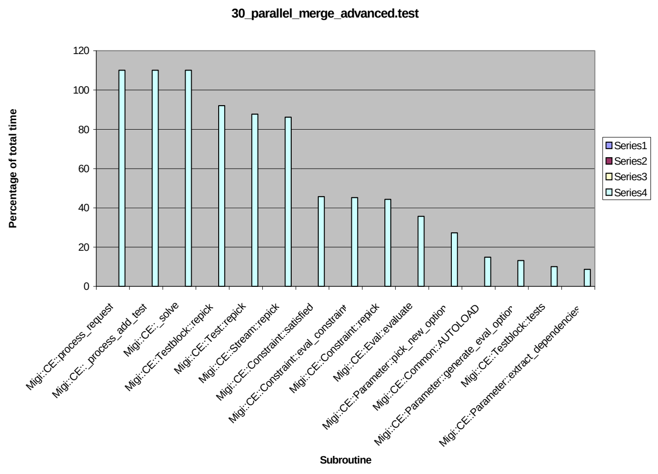
12) Using short circuit logic:

use the logical || operator, Perl will use the first true value it comes across, in order, from left to right. The moment it finds a valid value, it doesn't bother processing any of the other values.

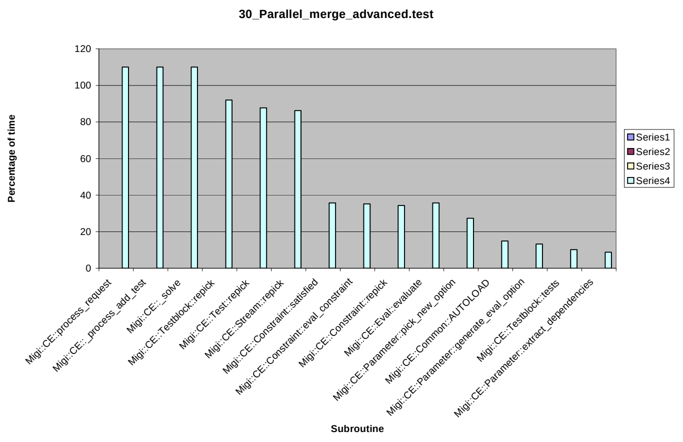
13) cache the array list, and then return the cached copy instead of re-creating the array all the time.

V. Experimental Results

Performance Analysis is to identify the Non-optimized Perl subroutines in the Constraint Engine Part. By Running the Command Lines for different tests and analyzing the Profile data, the following is the Result



Optimized Constraint.pm in CE



References

[1] When Perl is not quite Fast enough” by Nicholas Clark
Link: http://www.ccl4.org/~nick/P/Fast_Enough/

[2] Data Structures and Algorithms with examples in Perl by Jon Jacky.
Link: http://staff.washington.edu/jon/dsa-perl/dsa-perl.html