BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE OPTIMIZATION OF TESTING AUTOMATION FRAMEWORK BASED ON BEAKERLIB

OPTIMALIZACE VÝKONU AUTOMATIZOVANÉ TESTOVACÍ PLATFORMY ZALOŽENÉ NA BEAKERLIBU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAKUB HEGER**
AUTOR PRÁCE

SUPERVISOR **Mgr. Bc. HANA PLUHÁČKOVÁ**
VEDOUCÍ PRÁCE

BRNO 2017

Bachelor's Thesis Specification/20237/2016/xheger00

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems | Academic year 2016/2017

Bachelor's Thesis Specification

For: **Heger Jakub**
Branch of study: Information Technology
Title: **Performance Optimization of Testing Automation Framework Based on Beakerlib**
Category: Software analysis and testing

Instructions for project work:

1. Study how BeakerLib (integration test library) works.
2. Analyze performance of BeakerLib, design the metric of performance which would be optimized and identify the functional areas of BeakerLib and chosen harness to optimize performance (based on architectural review of the system, code review, code performance analysis).
3. Prepare and describe test set and environment for performance measurement.
4. Perform initial base line measurements, select at least one optimization and implement this optimization, e.g., by modification of BeakerLib code.
5. Check implemented optimization and discuss results.

Basic references:

- according to the instruction of the supervisor

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Pluháčková Hana, Mgr. Bc.**, DITS FIT BUT
Beginning of work: November 1, 2016
Date of delivery: May 17, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2
L.S.

Petr Hanáček
Associate Professor and Head of Department

Abstract
Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt
Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords
Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova
Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference
HEGER, Jakub. *Performance Optimization of Testing Automation Framework Based on Beakerlib*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Pluháčková Hana.

Performance Optimization of Testing Automation Framework Based on Beakerlib

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Mgr. Bc. Hany Pluháčkové. Další informace mi poskytl Mgr. David Kutálek. ??? Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .

Jakub Heger

May 15, 2017

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

Chapter 1

Introduction

Focus of this thesis is a performance optimization of Red Hat's BeakerLib library, particularly its Journal feature. <expand>

The thesis is structured in a following way: chapter 2 introduces projects relevant to BeakerLib and its testing environment. Chapter 3 explains more in-depth how BeakerLib works, with focus on its Journal feature and analysis of its performance. In the chapter 4 possible optimizations are discussed and chapter 5 focuses on implementation of proposed solutions. The chapter 6 then describes how was performance measured and in what environment. <expand>

Chapter 2

Relevant projects

This this chapter describes BeakerLib and projects relevant to it. Nejprve je popsana knihovna jako takova, nasledujici sekce se venuje Beaker coz je system from which BeakrLib originally a posledni sekce popisuje test harnessy

2.1 BeakerLib

BeakerLib is a Linux shell-level integration testing library, providing convenience functions which simplify writing, running and analysis of integration and blackbox tests. [12] It is developed and maintained by Red Hat and operates under GNU General Public License. Main features of BeakerLib include:

- Journal - uniform logging mechanism (logs and results saved in flexible XML format, easy to compare results and generate reports)
- Phases - logical grouping of test actions, clear separation of setup / test / cleanup
- Asserts - common checks affecting the overall results of individual phases (checking for exit codes, file existence and content...)
- Helpers - convenience functions for common operations such as managing system services, backup and restore of files and more

This thesis focuses on BeakerLib Journal and problem it causes with long tests. ??? Which will be described in chapter ...

2.2 Beaker

Beaker[11] is a full stack software and hardware integration testing system, with the ability to manage a globally distributed network of test labs. It is Red Hat community project under GNU General Public License version 2.

Main functionality includes management of hardware inventory, on which Beaker can install wide variety of operating systems from Red Hat Linux family. Another notable part is Task library which contains rpm packages of individual tests which can be run on provided machines. Users then can specify which hardware they require with which OS and tests they want to run on it through either command-line tools or web interface both of which are part of Beaker install package. If Beaker meets given criteria in its inventory it installs Test harness to which it gives list of tests to be run. After Test Harness finishes running the tests, results are sent back to Beaker where they are stored for specified period of time.

2.2.1 beaker-wizard

Part of Beaker package. Interactive command-line tool which automates creation of BeakerLib tests. Using predefined or user-defined templates it creates all files that are needed to run BeakerLib test. ??? Common use cases

2.3 Test Harness

Test harness is a software framework that automates test execution. It contains tests to be run, executes them and reports results. <expand>

Beaker's harnesses prepare provided machine for BeakerLib by setting environmental variables to proper values, and then running the tests.

2.3.1 Beah harness

Beah [10] is a default Beaker harness . <expand>

2.3.2 Restraint harness

Restraint [6] is an alternative Beaker harness which can, unlike Beah, run with Beaker or standalone without it. <expand>

2.4 Projects' relation

Relation between Beaker, Harness and BeakerLib is shown in figure 2.1. In this example user submits Beaker job containing three tests and hardware/software requirements for a machine the tests should run on. After Beaker reserves it, it installs Harness which successively executes each test and uploads their results back to Beaker when user can access them.

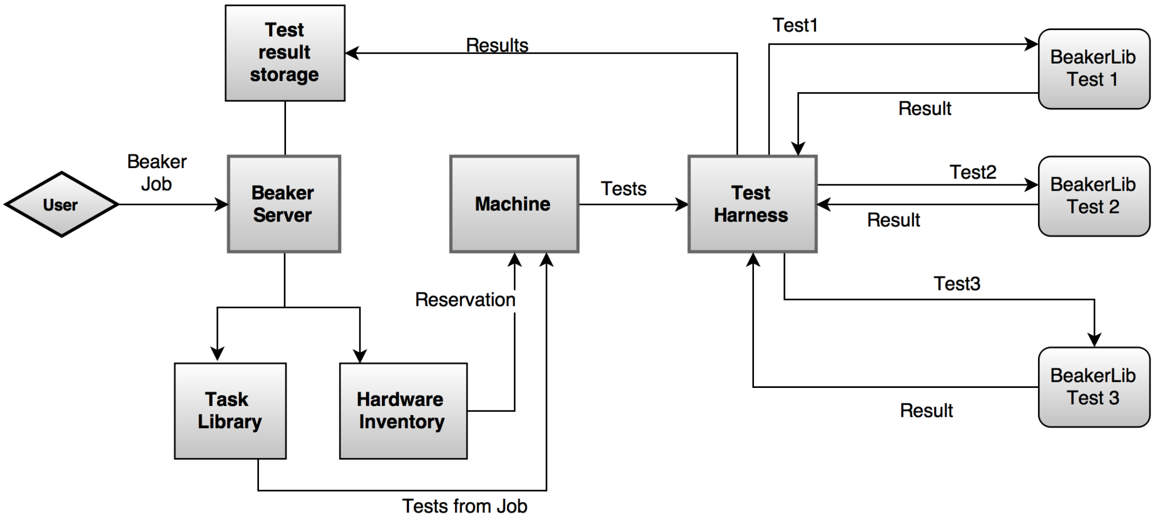

Figure 2.1: Beaker relation to BeakerLib

Chapter 3

BeakerLib

This chapter takes a closer look on inner workings of BeakerLib, with focus on Journal feature and performance issues it suffers from.

3.1 Important functions

As stated earlier BeakerLib is shell-level library with functions that make writing and running tests easier as well as examining their results. BeakerLib adds testing functions to `shell` functionality, so user can combine normal `shell` commands and constructions with helping functions which can make writing tests and examining their results easier. There is close to 80 of these functions (also known as **rlCommands**), description of most used ones follows:

- **rlRun** - First argument of this function is any shell command, which is executed by rlRun. Second is an expected exit code of first argument, it can contain one or more codes. Third argument is a comment. BeakerLib logs `FAIL` or `PASS` if expected exit code differs or not from actual one respectively along with comment. This is most used and important function.

- **rlPass** - Manual assertion and logging of `PASS`. Useful when in combination with if statement which user doesn't want to appear in logs but still wants to log its result. Reciprocal function **rlFail** exists as well.

- **rlAssertExists** - Asserts whether file given as a first argument exists.

- **rlAssertGrep** - Function logs `PASS` when pattern given as first argument matches in a file which is given a second argument. Optional flags are passed to `grep` and behave the same way.

- **rlAssertRpm** - Function asserts `PASS` when package given as first argument is installed. Optional arguments allow specifying particular version, release or arch of the package.

- **rlAssertDiffer** - Asserts whether two files given as argument differ in their content.

- **rlJournalStart** - This function is used at the start of each test. It is essential for proper run of the test as it initializes BeakerLib's outputs, which described later in this chapter. Reciprocal function **rlJournalEnd** must be called at the end of the test.

- **rlPhaseStart** - This function starts user-defined phase. Function takes two arguments, first one is a type of phase, second one is a name. Phase must be ended by calling **rlPhaseEnd**. Phases are more closely explained in the next section.

3.2 Phases

BeakerLib divides tests into logical groups called Phases. There are three predefined types of phases:

- Setup - Preparing conditions for the test (such as creating temporary files, starting needed system services and so on), started by calling **rlPhaseStartSetup**.
- Test - Main phase for testing, started by calling **rlPhaseStartTest**.
- Cleanup - Reverting changes made by the test, started by calling **rlPhaseStartCleanup**.

Apart from predefined Phases, user can also define own phases by calling **rlJournal-Start** function. First argument of the function is a one of two types phase can have:

- WARN - if any **rlCommand** in phase of this type fails, whole phase will result in Warning state.
- FAIL - similar to previous type however this time resulting in Failed state.

Basic phases Setup and Cleanup are WARN type, Test phase is a FAIL type.

The result of the whole test is the same as the worst result of any phase in the order: Failed, Warning, Passed. Asserts must not be used outside of phases, if such a case occurs, new

This division helps with examining the result of test as it shows which phase, if any, causes fail in BeakerLib's output. example test 3.1 shows how basic BeakerLib test looks.

```
# Include Beaker environment
. /usr/bin/rhts-environment.sh || exit 1
. /usr/share/beakerlib/beakerlib.sh || exit 1
# Start of Journal
rlJournalStart
    # Start of Setup Phase, creating temp directory where test will take place
    rlPhaseStartSetup
        rlAssertRpm $PACKAGE
        rlRun "TmpDir=\$(mktemp -d)" 0 "Creating tmp directory"
        rlRun "pushd $TmpDir"
    rlPhaseEnd
   # Start of Test Phase, testing touch and ls commands
    rlPhaseStartTest
        rlRun "touch foo" 0 "Creating the foo test file"
        rlAssertExists "foo"
        rlRun "ls -l foo" 0 "Listing the foo test file"
    rlPhaseEnd
   # Statr of Cleanup phase, temp directory is deleted
    rlPhaseStartCleanup
        rlRun "popd"
        rlRun "rm -r $TmpDir" 0 "Removing tmp directory"
    rlPhaseEnd
rlJournalPrint
rlJournalEnd
```

Listing 3.1: BeakerLib basic test example

3.3 BeakerLib's output

BeakerLib produces three kinds of outputs, two file formats and a console one which are described in this section. Files are saved into a directory created for each individual test. If the test is run locally, temporary directory is created on system with `mktepm` command. If run on Beaker unique **TESTID** is generated for each test. This **TESTID** serves as a name for test directory as well as identifier which Beaker later uses when connecting test results with correct test.

3.3.1 journal.txt

journal.txt is a plain text file with human readable record of test's progress. After end of each phase, copy of the file is sent to Beaker for storage. Snippet of *journal.txt* generated by Example test 3.1 is shown in 3.2.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Checking for the presence of bash rpm
:: [   LOG    ] :: Package versions:
:: [   LOG    ] ::   bash-4.3.43-4.fc25.x86_64
:: [   PASS   ] :: Creating tmp directory (Expected 0, got 0)
:: [   PASS   ] :: Command 'pushd /tmp/tmp.oawa0RcDNI' (Expected 0, got 0)
:: [   LOG    ] :: Duration: 1s
:: [   LOG    ] :: Assertions: 3 good, 0 bad
:: [   PASS   ] :: RESULT: Setup

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Test
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Creating the foo test file (Expected 0, got 0)
:: [   PASS   ] :: File foo should exist
:: [   PASS   ] :: Listing the foo test file (Expected 0, got 0)
:: [   LOG    ] :: Duration: 0s
:: [   LOG    ] :: Assertions: 3 good, 0 bad
:: [   PASS   ] :: RESULT: Test

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Cleanup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Command 'popd' (Expected 0, got 0)
:: [   PASS   ] :: Removing tmp directory (Expected 0, got 0)
:: [   LOG    ] :: Duration: 0s
:: [   LOG    ] :: Assertions: 2 good, 0 bad
:: [   PASS   ] :: RESULT: Cleanup

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: /examples/basic/Sanity/basic-test
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   LOG    ] :: Phases: 3 good, 0 bad
:: [   PASS   ] :: RESULT: /examples/basic/Sanity/basic-test
```

Listing 3.2: Example of journal.txt

3.3.2 Console output

If the executed test is connected to an `interactive shell` similar, human-readable, output to the *journal.txt* is also printed to console's standard output (`stdout`). Apart from

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Test
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [  BEGIN   ] :: Creating the foo test file :: actually running 'touch foo'
:: [   PASS   ] :: Creating the foo test file (Expected 0, got 0)
:: [   PASS   ] :: File foo should exist
:: [  BEGIN   ] :: Listing the foo test file :: actually running 'ls -l foo'
-rw-rw-r--. 1 jheger jheger 0 May 15 03:20 foo
:: [   PASS   ] :: Listing the foo test file (Expected 0, got 0)
```

Figure 3.1: Snippet from console's output

journal.txt's content console's output is complemented by executed command's output. Also `shell's` output is colored for increased readability. Figure 3.1 shows snippet of such output.

3.3.3 TESTOUT.log

If the executed test is not connected to an `interactive shell`, the same text as in Console's output is printed into file *TESTOUT.log*. This is mostly the case when executing a test remotely (for example in Beaker), where it is not possible to see the Console output.

3.3.4 journal.xml

Last output is a an XML[1] file. This document is stripped off of executed commands' own output, but core information (such as which commands were executed, whether they passed or failed and so on) is kept. Also metadata about the test run (time of execution, which component was tested and more) as well as information about the hardware and software test was run on are added. *journal.xml* is sent back to Beaker same as *journal.txt* where it is available for further processing by automated tools. It also serves as a source of information about current state of the test during its execution, for example whether there is currently an open phase or how many failed tests or phases there are so far.

3.4 Source files

This section describes a few of BeakerLib's source files, relevant to this thesis.

- *beakerlib.sh* - Starting point of every tests. It is `sourced` at the beginning of each test and in turn `sources` all other BeakerLib files.
- *testing.sh* - Contains definitions of the most used **rlCommands** as well as some internal functions.
- *journal.sh* - Provides bash-side Journaling functionality. Functions from this file process information about what to log and relay them to *journalling.py*.
- *journalling.py* - Python script responsible for creating most of BeakerLib's outputs. It also creates and modifies *journal.xml* file.

[1] eXtensible Markup Language

3.5 Analysis of slow performance

It was reported that BeakerLib suffers performance problems when running long tests. Time of processing each **rlCommand** grew longer after many (several hundreds and more) were used. Analysis of library was problematic due to lack of documentation, complex structure and uncommented code, however thorough investigation of the source code indicated that problem lies with generating *journal.xml*.

Python script *journalling.py* is called after each **rlCommand** to log its result into *journal.xml*. This isn't big problem with small test as the *journal.xml* file takes up only a few kilobytes, however when the file takes up dozens or hundreds of kilobytes, repeated loading the file from disk, parsing, adding a line of log and then saving the file back to the disk adds significantly more load to CPU[2]. Running larger tests therefore becomes quite time consuming and considerably slows down testing as a whole.

This has been determined as the main focus of the thesis since it probably is the most significant performance bottleneck. The next chapter describes proposed solutions with their pros and cons.

Figure 3.2 illustrates simplified version of how **rlRun** propagates through different functions from BeakerLib files (which are `sourced` at the time test execution, depicted by rounded rectangles) and how it is logged into the Journal. Similar operation is performed for every **rlCommand** in a test.

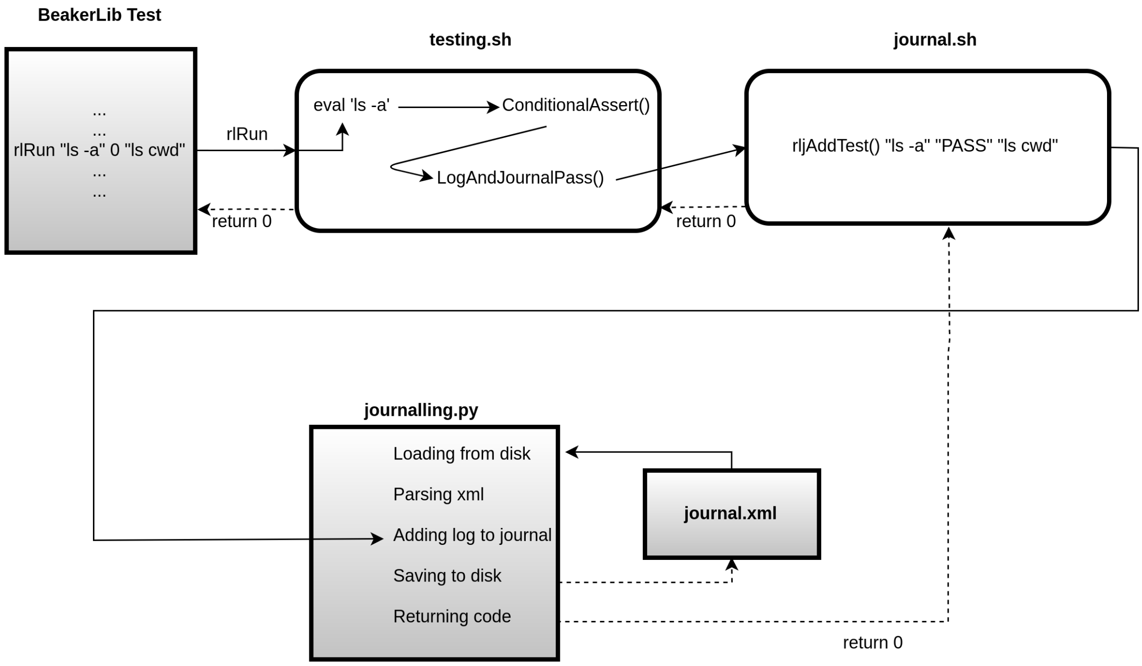

Figure 3.2: Logging of rlRun to Journal

[2]Central Processing Unit

Chapter 4

Solution of Journaling problem

This chapter describes proposed solutions to optimize Journaling problem. Sections in this chapter provide possible solutions to the Journaling problem. Besides explaining the principle of each solution, the sections also discuss their advantages, disadvantages and potential issues.

4.1 Change of xml parser

XML parser is a program which can turn XML document into structured object in RAM[1]. Depending on implementation of the parser, that object is then easier to access by the program as it may provide methods to navigate the object and search it or potentially modify.

Parsing of XML in BeakerLib is done by *journalling.py* script by `Python` module `xml.dom.minidom`.

`xml.dom.minidom` is a native part of `Python` from version 2.0 and provides minimal implementation of the DOM[2] interface, with an API[3] similar to that in other languages. [8]

I decided to change parser to different one, to measure whether it will provide better performance. Because of reasons of backward compatibility with RHEL 5[4] which needs to be supported by BeakerLib, the choice of XML parsers was limited to native modules of `Python 2.4.3`. Two additional XML parsers were present for given version.

- lxml - The `lxml` XML toolkit is a Pythonic binding for the C libraries libxml2 and libxslt. It combines the speed and XML feature completeness of these libraries with the simplicity of a native `Python` API. [2]

  It works similarly to `xml.dom.minidom` in the way that when reading XML object from a file, it will read it whole, builds an object out of it and provides methods for the object to allow access to it.

- xml.sax [9] - `xml.sax` originated as a parser for `Java`[4]. In `Python` it was released with version 2.0. It differs from `xml.dom.minidom` and `lxml` where the two mentioned parsers work with a whole XML file, `xml.sax` emits events as it goes step by step

[1] Random Access Memory
[2] Document Object Model
[3] Application Programming Interface
[4] Red Hat Enterprise Linux 5

through the file[7]. Using this approach means less memory has to be allocated for XML handling and therefore makes it ideal when working with very large amount of XML data.

I decided to implement `lxml` parser as it is supposed to be faster and less demanding on memory than `xml.dom.minidom`[3], while keeping its intuitive interface.

4.2 Change in calling journalling.py

Next proposition to make BeakerLib faster is in a way *journalling.py* is called. The assumption being that repeated parsing of XML document slows BeakerLib the most, reducing the number of times it was parsed was then the highest priority.

4.2.1 Queue file solution

First solution is to create a new, temporary **queue file**, which will act as a kind of buffer. **rlCommands** will behave as before apart from creating BeakerLib journals, but instead they will write message into the **queue file**. This file will be read and processed only when necessary, that is at the end each phase, when journals are sent to Beaker.

Disadvantages

The way BeakerLib is designed now it in most cases expects some form of return value from *journalling.py* immediately after adding a log to a journal. Performed logging either returns code indicating success of failure or `string` with information about the current state of test. This presents problem as there is no way how to communicate back these information when parsing is postponed.

4.2.2 Daemon-like solution

Second solution is to rewrite *journalling.py* script to have daemon-like behavior. That is to have it run as its own process throughout the whole time of test execution. The XML object will be stored in memory, and parsed as whole only at the beginning of journal creation and in case of restarting the test run (which happen when machine is rebooted).

This way BeakerLib can receive response about current test state immediately while still keeping CPU load minimal. Daemon-like solution however brings different obstacles.

Disadvantages

An independent, potentially long running process daemon is more prone to unplanned events such as unexpected exit. This must be addressed by both daemon (to exit as safely as possible) and by the rest of BeakerLib (to detect that daemon is no longer running and to behave accordingly).

Communication

Inter-process communication between running test and daemon has to be created for test to inform which **rlCommand** is supposed to be logged and for daemon to respond with current state of XML document. This two-way communication must be synchronous to assure

BeakerLib and daemon process their respective messages in correct order. I considered following options:

- Unix sockets - <expand>
- Named pipes - Named pipes are device files. They allow inter-process communication by reading it and writing into is as if regular file, however under normal circumstances the read/write is a blocking operation[5]. This means if one process opens pipe for reading, it will hang there until another process opens the pipe for writing. This feature can be used for synchronization of communication between processes.

I chose to implement communication through Named pipes because synchronization issue is taken care of because of the way Named pipes are designed.

Chapter 5

Implementation of proposed solutions

This chapter describes how the proposed solutions were implemented. Each solution has its own section that describes implementation details and obstacles that were found and had to be solved during the implementation. During changing of parsers I discovered and fixed few bugs present in current implementation of *journalling.py*.

5.1 Change of xml parser

5.1.1 Difference in parsers

As mentioned before I chose to change original XML parser to `lxml`. Only changes in code were in file *journalling.py* as it is only part of BeakerLib that directly works with journal's XML object. Most of the changes consisted in changing `xml.dom.minidom's` method for creating new XML element and assigning value into it. Biggest difference between given parsers is that `lxml` does not provide many helping methods as `xml.dom.minidom` does. For example in `lxml` there is no method `getElementsByTagName()` to search XML object by a tag name. Instead `lxml` supports `xpath` [13] syntax for searching the object. xpath[1] is part of XSLT[2] standard. It be can used to navigate through elements and attributes in an XML document.

Another example of difference is an approach for accessing element's children. While `xml.dom.minidom` has dedicated methods and attributes such as `hasChildNodes()` which returns `bool` value or `childNodes` which is a iterable attribute of element's children, `lxml` has more low level implementation. It treats elements as python lists so `hasChildNodes()` can be replaced with simple `len(element) != 0`.

Because preliminary performance measurement showed faster test execution with `lxml`, I decided to implement the rest of the proposed solutions with this parser.

[1]XML Path Language
[2]eXtensible Stylesheet Language Transformations

5.2 Queue file solution

This section deals with implementation of **queue file** solution. It is divided into subsections that discuss files I designed or changed during implementation.

5.2.1 Queue file

Queue file was designed in a way so it was simple to implement, in a human readable format for potential test debugging and easy to extend by new, future functions that will work with it. It is a plain text file, each line containing one buffered command for `Python` script to process later, on demand.

5.2.2 journal.sh

Creation of **queue file**, by using `touch` command, was added to function **rlJournalStart()** which initializes Journaling functionality. It also `exports` new variable BEAKERLIB_QUEUE, with **queue file's** path, into test's environment so `Python` script *od_journalling.py*, can later access it.

Original calling of *journalling.py*, which is a main functionality of *journal.sh*, was replaced in one of two ways:

- Delayed calling - New function **rljPrintToQueue()** takes all arguments that were originally meant for *journalling.py* and instead prints them into **queue file**, where it will be processed by *od_journalling.py* later during execution of the test. This concerns functions which do not necessary require response about current test state from *journal.xml*. Namely functions: rlJournalPrint(), rljAddTest(), rljAddMetric(), rljAddMessage(), rljRpmLog()
- Immediate calling of *od_journalling.py* - Virtually the same as the original solution. These functions require immediate response. Using this way of calling won't save on any CPU load (in fact the the load will be slightly higher than before because of operations related to **queue file** processing), however in typical BeakerLib test these functions are in minority compared to previous type of calling. Functions and the response they require are:
  - rlJournalStart() - requires confirmation that journal was initiated successfully
  - rlJournalPrintText() - requires *journal.txt* which is generated from current *journal.xml*
  - rlGetTestState() - requires number of failed asserts in the test so far
  - rlGetPhaseState() - requires number of failed phases in the test so far
  - rljAddPhase() - requires immediate print
  - rljClosePhase() - requires result of closed phase, to send it to Beaker along with Journal
  - rlJournalStart()

Function **rljAddTest()** is the cause of the most calls of *journalling.py* in original solution, therefore had the highest need to be moved into group of functions with Delayed calling. However it does require knowledge of current state of the test. That being situation when Assert (rlCommand using **rljAddTest()** for Journaling) is used outside of a phase, such information is held only in current *journal.xml*. To solve this problem functionality of **rljAddTest()** had to moved into *od_journalling.py* script, discussed in the next subsection.

Apart from printing to **queue file**, **rljPrintToQueue** also has to escape given arguments. This done because firstly some of the arguments originating from user may contain newline character which would break the„one buffered command per line" rule in **queue file's** format and secondly so *od_journalling.py* may parse it with `optparse` module. Escaping is done with `printf` bash builtin[1], specifically its `%q` option which causes `printf` to output in `shell-quoted` format.

5.2.3 od_journalling.py

File *od_journalling.py* originated from *journalling.py* but it differs in some ways.

Now when it is called, it first parses current *journal.xml* and then calls new method `updateXML()` with parsed XML object as an argument. This method opens **queue file** and finds last line it accessed in previous call. From there it reads buffered lines, which are decoded from escaped and modifies the XML object accordingly.When it reaches end of file, it makes a mark for future readings and returns to the original call coming from one of the *journal.sh's* Immediate calling functions. After modification from that function it generates response and returns it to *journal.sh*. Exceptions to this behavior are calls from rlJournalStart(), which doesn't access textbfqueue file but only initializes XML object and returns exit code whose value depends on how was initialization successful and rlJournalEnd() which makes sure every buffered command was processed as it is an exit point from the test.

5.2.4 Problems with implementation

Main goal of this solution was to reduce number of times *journal.xml* is parsed, by delaying as many Journaling operations as possible, while keeping BeakerLib's outputs the same. The way BeakerLib is designed now it is not possible, because some information is always lost when operations are delayed. In case of this implementation I was able to keep *journal.xml*, and therefore *journal.txt* as well, the same as with original solution, however at the price Console's output which is now missing some information usually given by functions from Delayed calling category.

Solving this issue would require larger changes to BeakerLib's design which I decided not to implement for now so **Queue file** solution remains only as a proof of concept.

5.3 Daemon-like solution

5.4 testing that my solution is the same

Chapter 6

Performance measuring

<Definition of performance measuring>

For performance measuring of BeakerLib I chose two kinds of tests in in two kinds of testing environments.

6.1 Tests

6.1.1 Artificial tests

First type of tests are artificial tests created by me with `beaker-wizard` tool to specifically target and measure performance of journaling modifications I made. They consist mostly of **rlCommands** that directly work with `journalling.py`. For example commands **rlLog** or **rlPhaseStart** and **rlPhaseEnd**. This way we can observe clear difference in performance without being affected by operations unrelated to journaling (executing actions that verify functionality of components in real tests).

<description of artificial tests with links to Appendix>

6.1.2 Real tests

Second type are real tests used in Red Hat. These are examples of tests that have been reported to have bad performance with BeakerLib so I am testing them to see if my modifications have real life impact on performance.

<description of real tests>

6.2 Testing Environment

6.2.1 Local

First environment is local laptop for convenience and speed of execution. Tests were run directly, without any harness and with these technical specifications.

<table of tech specs>

6.2.2 Remote in beaker

Second round of testing was done to emulate real testing conditions and to verify that changes made to BeakerLib do not break functionality outside of controlled environment. Tests were run with the default test harness Beah. <table of tech specs>

6.3 Measured Values

6.4 Baseline measurements

<results>

6.5 Implemented optimizations

<results>

Chapter 7

Conclusion

Recap of results
Future work

Bibliography

[1] bash_builtins(1) General Commands Manual. April 2004.

[2] Behnel, S.: *lxml documentation*. [Online; visited 15.05.2017].
Retrieved from: http://lxml.de/index.html

[3] Behnel, S.: *lxml performance*. [Online; visited 15.05.2017].
Retrieved from: http://lxml.de/performance.html

[4] Brownell, D.: *Sax2*. O'Reilly Media. 2002. ISBN 0596002378.

[5] Goldt, S.: *Blocking Actions on a FIFO*. [Online; visited 16.05.2017].
Retrieved from:
http://www.tldp.org/LDP/lpg/node19.html#SECTION00734000000000000000

[6] Peck, B.; Callaghan, D.; Bastian, J.; et al.: *Restraint documentation*. [Online; visited 15.05.2017].
Retrieved from: http://restraint.readthedocs.io/en/latest/

[7] Python Software Foundation: *Sax*. [Online; visited 15.05.2017].
Retrieved from: https://wiki.python.org/moin/Sax

[8] Python Software Foundation: *xml.dom.minidom documentation*. [Online; visited 15.05.2017].
Retrieved from: https://docs.python.org/2/library/xml.dom.minidom.html

[9] Python Software Foundation: *xml.sax documentation*. [Online; visited 15.05.2017].
Retrieved from: https://docs.python.org/2/library/xml.sax.html

[10] „Red Hat Inc.“: *Beah documentation*. [Online; visited 14.05.2017].
Retrieved from: https://beah.readthedocs.io/en/latest/

[11] „Red Hat Inc.“: *Beaker documentation*. [Online; visited 14.05.2017].
Retrieved from: https://beaker-project.org/

[12] „Red Hat Inc.“: *BeakerLib GitHub wiki man page*. [Online; visited 15.05.2017].
Retrieved from: https://github.com/beakerlib/beakerlib/wiki/man

[13] W3Schools: *XPath Tutorial*. [Online; visited 15.05.2017].
Retrieved from: https://www.w3schools.com/xml/xpath_intro.asp

Appendices

List of Appendices