KTH Information and
Communication Technology

Performance benchmarking using real world applications

An attempt to create a benchmarking tool for measuring software performance in realistic scenarios.

MATTIAS JANSSON, KARL JOHANSSON

Bachelor Thesis at ICT
Supervisor: Frej Drejhammar (SICS)
Examiner: Christian Schulte (School of ICT)

TRITA-ICT-EX-2012:144

Abstract

A generic tool for creating and running benchmark tests on existing software was designed and created. The purpose of the tool is to simplify testing and tuning during the development of an Erlang JIT compiler. The resulting tool is implemented in Python and is based on a set of jobs running sequentially or in parallel. This gives a simple way to define initialization, parallel runs and cleanup of a benchmark run. This is achieved by inter-job dependencies. Logging features are built into the tool for creating output data. For data presentation, a simple plotting tool was created specifically for reading the output format of the benchmark tool and plotting with gnuplot. Another tool for running benchmarks on remote computers was added as well. The tools give a quick way of creating and running benchmarks and also gathering data and presenting them.

All measuring is performed on application level only and no profiling functionality is added. Therefore, the tool should not be seen as a substitute for profiling but as a complement to it.

Referat

Prestandatester med hjälp av verkliga applikationer

Ett generiskt verktyg för skapande och körning av benchmarktester på existerande programvara framtogs. Syftet med verktyget är att förenkla testning och optimering vid utveckling av en JIT-kompilator för Erlang. Verktyget är implementerat i Python och är baserat på en uppsättning jobb som kan köras sekvensiellt eller parallellt. Detta ger ett enkelt sätt att definiera initiering, parallell exekvering och avslut av benchmarktester. Detta möjliggjordes med hjälp av beroenden mellan jobb. Loggningsfunktionalitet är tillgänglig i verktyget för skapande av utdata. För presentation skapades ett enkelt verktyg specifikt för läsning och uppritande av data. Ett annat verktyg implementerades för fjärrstyrd körning av tester. Verktygen ger ett snabbt sätt att skapa och köra benchmarktester samt lagring och presentation av data.

All mätning görs på applikationsnivå och ingen profileringsfunktionalitet är tillagd. Därför skall verktyget ej ses som ett substitut för profilering utan snarare som ett komplement.

Contents

Chapter 1

Introduction

An important aspect of software development is efficiency. To make software perform better, a good practise is to continously measure performance differences between versions, settings etc.

Performance measurements can be done by constructing a synthetic benchmark scenario where a specific task is performed while performance is measured, similarly to the method described in "UTS: A Portable Benchmark for Erlang/OTP" by Östberg [4]. However, using that method for evaluating a compiler or execution environment raises an important question: do the results reflect the performance of a "real world" application?

In the development of a JIT compiler for Erlang there was a need for performing performance benchmarks in order to correctly tune the compiler.

This thesis is an attempt to create a benchmark utility suite where performance of any kind of application can be measured. The goal is to easily be able to create a test scenario out of a real application and run it with many different setups. The tool will hopefully be of benefit during development of compilers, execution environments, software libraries or other software. This could produce data that helps in creating software that is efficient not only in synthetic tests but also in practice.

Chapter 2

Background

2.1 Problem

2.1.1 Benchmarking tool

The benchmark tool is designed with the following criteria in mind:

General purpose functionality
A more general approach allows us to reuse the tool for other benchmarking projects that require slightly more complex testing.

Measuring execution time and memory usage in multiple iterations
An important part of benchmarking is to measure execution according to some attribute of interest. A typical usage is measuring how long it takes to run a piece of code or how much memory is needed. Focusing on execution time, the tool needs to be more specific and provide time spent with IO, on the kernel level and so on.

Data storage and presentation
Handling benchmark results might require different techniques depending on the user's needs. For example, results to be used in a presentation might need automated formating. There might also be situations where the results are to be used by other scripts or programs. In that case, proper structure and simplicity is preferred. The tool needs to either be able to cater to these needs or be easily extended.

Automatic download and installation of test application
Manually installing programs and libraries needed for testing quickly becomes tedious. Programs must therefore be downloaded and installed automatically if needed.

2.2 Python

The core of the benchmark tool is written exclusively in Python. Therefore, a small overview of the language and related issues are brought up in this section.

2.2.1 Brief overview of Python

Python is an interpreted, general purpose script language, covering multiple paradigms [3]. Borrowing style from C, the language attempts to be as syntactically uncomplicated as possible and familiar to most programmers [9]. The similarities end quickly however as Python is at the opposite extreme to the static languages. The dynamic type system is strict [3]. Unless objects are initiated, variable types are implied. Variables are only defined, never declared. Furthermore, classes can be defined at runtime. The benchmark tool makes heavy use of this.

2.2.2 Backwards compatibility issues

Because of backwards compability issues, the decision had to be made on which version to use. During the initial design there was an urge to make heavy use of external libraries, thus the older version (Python 2.7) was chosen in favor over Python 3.

When Python 3 was released, backwards compatibility was removed in order to clean up the core [12].

2.3 Measuring process resource usage

In order to obtain reliable data from measurements, the resource usage data must be both of high precision and detail. For example, measuring overall execution time might not be sufficient without details about how the time is spent (waiting for system calls, memory swapping etc).

In the POSIX standard there is a system call specified called *getrusage* [10]. This system call can be used for obtaining a variety of resource usage details about either the calling process or child processes that have terminated and have been waited for. The *getrusage* system call provides information that might be needed in this benchmarking tool.

The system call is available not only as a C system call but also other languages such as Python [5]. The high detail, operating system integration and high availability makes the call a good way to obtain resource usage information.

2.4 Erlang

The purpose of the project was to provide a tool that can be used to test and tune a compiler for the Erlang programming language. The examples in Chapter 4

are therefore focused on Erlang. This section will give a very brief presentation of Erlang as well as OTP.

Erlang is a functional programming language especially designed for concurrency, distribution and fault tolerance. According to *Erlang FAQ*, Erlang is particularly suited for "distributed, reliable, soft real-time concurrent systems", such as telecommunication systems, Internet server applications, telecommunication applications and database applications. [2]

Erlang FAQ mentions that Ericsson is "the largest user of Erlang", but it is used by many others for different purposes. For example, Erlang is used for VoIP comunication, web services, digital payment system, social networking and more. [2]

2.4.1 OTP - Open Telecom Platform

OTP (Open Telecom Platform) is a collection of libraries for Erlang provided by Ericsson [2]. It is very commonly used as the main library collection for Erlang application development. Despite what the name suggests, OTP is not only targeted to the telecoms sector. It is distributed with a wide variety of tools and libraries such as WWW server, database tools, analysis tools and much more.

In the examples in this report, OTP will be used for running all Erlang applications.

2.5 Statistical formulas used

The following formulas are used to calculate results:

Mean average:

$$\mu = \frac{\sum_{i=0}^{n} X_i}{n}$$

Standard deviation:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n} (X_i - \mu)^2}{n}}$$

Coefficient of variation (referred to as cov in this thesis) :

$$c_v = \frac{\sigma}{\mu}$$

These are used by our logging module described in Section 3.4.

Chapter 3

The benchmark tool in detail

This chapter begins with a basic outline of our design of tests as well as the concrete implementation in Section 3.1. In Section 3.2, how to configure tests are described. The design and implementation of remote testing is presented in Section 3.3. Lastly, how to store tests results are covered in Section 3.4.

3.1 Executing tests

This section describes the modules our tool consists of and how tests are run. We begin by giving a structural overview of the modules before going into detail.

3.1.1 Structural Overview

Before focusing on the specific details, a brief overview is needed to see the full picture. A complete test uses two abstractions:

Job contains specific details about what is to be executed, logging, and precision and iteration constraints. A job runs a specific task, which does not have to be something measurable.

Executor coordinates execution of multiple jobs according to their dependencies.

A test suite contains one or more executors, which contain one or more jobs. This is shown in Figure 3.1. One needs to understand how these two abstraction layers function in order to get the most out of the tool.

3.1.2 Jobs

The configuration of a benchmark is defined by a set of jobs. Jobs may include anything from running a bash command to a script or a program.

For example, when benchmarking an application, the first job can define how to download and compile a specific version of the application. When finished, another

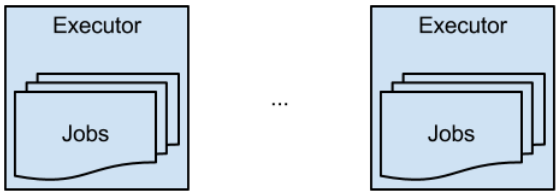

Figure 3.1. The two execution layers used for running tests.

job starts, runs the application and measures performance. When the measuring job terminates, a final job performs cleanup.

This way of structuring the benchmark process requires that jobs run in a certain order. Furthermore, there might also be a need for some jobs to run in parallel. Combining these types of jobs would require a configuration that offers high flexibility in job ordering and ensures that all jobs are run exactly when the user intends them to.

Jobs with fixed number of iterations

Since nondeterministic factors make testing unprecise, there may be a need to run tests multiple times. The user can specify how many iterations to use. If this is not specified, a job will run only once.

Jobs with precision constraints

Running a test multiple times can enhance the precision of measurements, but we cannot always be sure how many iterations are needed in order to get satisfying results. A type of job that executes a command under two constraints has therefore been implemented. The two constraints are the following:

- A maximum number of iterations.
- A precision boundary.

This job stops executing either when the maximum number of iterations has been reached or when the precision is deemed good enough. In order to specify what is considered good enough, we have chosen to let the user specify the coefficient of variation. All statistical methods used are presented in Section 2.5.

Like the job only using fixed iterations, this precision bound job has a maximum number of iterations to avoid nontermination. Unlike the fixed iterations job, this job will run a command a minimum of two times. The reasoning is the following: We cannot measure precision from a single run.

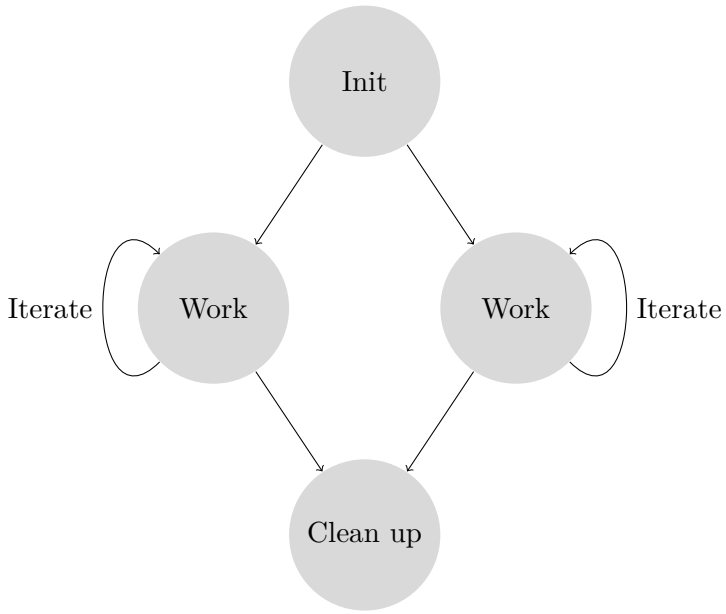

Figure 3.2. Example of jobs with dependencies. The ”work” jobs will execute as soon as the init job terminates. Lastly, when both ”work” jobs terminate, the ”cleanup” job will run.

3.1.3 Executors

A test suite is built of multiple jobs and these jobs need to be organized. This is the task of an executor.

Keeping track of jobs with a dependency graph

For each job, a list of zero or more other jobs can be listed as dependencies. The tool will check all dependencies and run all independent jobs first (jobs with zero dependencies). The jobs that do have dependencies will postpone execution until all jobs in it's dependency list are done. For example, two jobs with the same dependencies will be started roughly at the same time and run in parallel. This way, quite complex executions can be specified.

When an executor is initialized, a graph is constructed to help keep track of all the dependent jobs. This graph is called a dependency graph. Using the id numbers of every job, each is modelled with a node containing the id of all dependent jobs. Figure 3.2 shows what such a dependency graph might look like.

3.1.4 Implementation

We conclude the section on executing test with a more thorough description of how tests are executed. Here, the focus lies on implementation of execution.

Executor

Unlike jobs, executors are not run in parallel, only sequentially. When an executor starts, a reference to a signaling function (or notification function) is provided. This function is invoked whenever the executor has finished executing all jobs.

Jobs

Running a job is nontrivial, because we need to make use of parallelism (in order to enable setups such as in Figure 3.2); both by the use of fork and multithreading. The need for fork stems from using getrusage when measuring runtime and memory usage. Results from *getrusage* will include all child processes. Therefore, jobs are always started in different process groups.

The first step is to use fork and let the child process execute the command. The parent process needs to be notified when the child process finishes. When the child process terminates, the job is considered finished and other jobs who depend on it have to start. However, there might be other jobs scheduled to execute in parallel with the one created.

One solution is to install a signal handler [6]. After the use of fork, more jobs can be started in parallel if needed. When the child process has finished, the signal handler function is invoked.

Instead of a signal handler, multiple threads can be used. After forking, a thread starts and blocks execution, by using the system call *waitpid* [7]. Providing the correct arguments, waitpid blocks until the child process has finished running, and then notifiesp the executor. Due to the simplicity of threading in Python, the solution using threads was chosen to wait for child processes to finish. This is shown in Figure 3.3.

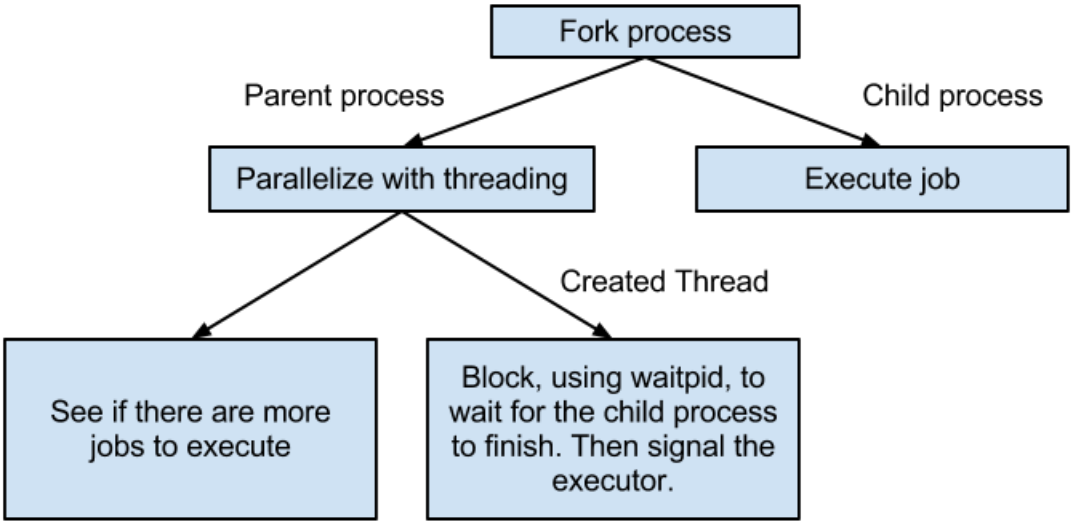

Figure 3.3. Parallelization strategy when executing jobs.

3.2 Configuration

This section explains how to set up configurations for each job. All commands are first briefly described in a summarized way, and the following sections describes them in more detail. How unknown and missing settings are dealt with is explained as well as the reason for the structure of the input files. What is needed to log results is explained in depth because it requires knowledge of quite a few settings. Examples of configuration files are found in Appendix A and Appendix B.

3.2.1 Structure of the input settings

Deciding on the structure of the input settings required consideration. We wanted to find a perfect balance, where users could create tests without much effort and knowledge of specific details. However, ongoing development needs to benefit from easily parsed input. First, the two extremes will be described. Then, the compromise used is presented

A structure that benefits the user

At one end of the spectrum, we have an input structure that benefits the user completely. The content would be language agnostic, avoiding any resemblance of a programming language. Input files containing multiple jobs would never have anything repeated and references to other jobs depends on a lot of implicit information or formating of the file.

For input files, a parser would have to be incorporated into the benchmarker. For the purposes of this tool, the inconvenience of implementing a parser overweights the benefits.

A structure that benefits the developers

At the other end of the spectrum, we have a structure for the input file that benefits the developers of the benchmark tool. The input would no longer be a simple text file, but Python code defining everything needed using object oriented semantics. Little would have to be done by the our tool other than to import the settings and use them. Furthermore, using this design allows advanced extensions and customizations done by the user.

Every user should not have to learn python in order to use our tool. This benchmark tool has nothing to do with python other than being constructed in it. Only catering to python developers would cripple the usability.

Finding a compromise

Incidentally, python's flexibility allowed us to cover most of the benefits from both extremes. No parsing is needed because the input settings are made out of python code. However, only pythonistas will notice this because everything is stored in nested lists and map structures (called dictionaries in python). There is no notion of variables, class or function definitions, nor other complex terms that might scare off unfamiliar users.

3.2.2 Handling settings that are missing or unknown

In an early development phase of the benchmark tool, it was decided that input settings that are unknown to the tool are set as environment variables.

This caused spelling errors or other internal issues to be very hard to track and correct. Working with a dynamic language like python, it was evident that requiring more strict and explicit input was a blessing for error handling regardless of the small cost of a larger and inconvienient configuration structure. Thus any unknown input settings will result in a fatal error. More time consuming and deceptive pitfalls are now avoided.

3.2.3 Summary of commands

A brief summary is given in Table 3.1, before they are described in more detail.

When a job is to be measured and the results need to be logged, specific settings are required. These are described in Table 3.2. Note that either all the settings in Table 3.2 have to be specified or none of them. Any other variation is not allowed.

Command	Section	Description
id	3.2.4	A unique nonnegative integer id for each job. This allows our jobs to have an easy to reference identity.
iterations	3.2.5	The number of iterations to run a command at a time. Also sets the maximum number of iterations allowed for jobs with precision constraints.
cov	3.2.6	the coefficient of variation (explained further in Section 2.5), which is used as a constraint whenever a value between 0.0 and 1.0 has been set.
dependencies	3.2.7	contains the identification of other jobs to be run before this one is to be run. if not specified, all previous jobs will be set as the dependencies for the current job.
command	3.2.8	The actual command that is to be run and measured. This is the only input required,
env	3.2.9	Contains environment variables to be used for a specific job. These variables disappear when other jobs are done due to the use of fork for each job.
logging	3.2.10	Provides multiple options for handling logging of results. The different logging settings are described in Table 3.2 .

Table 3.1. Brief overview of the input settings and which section that describes them in depth.

Command	Description
name	Sets a specific name to be used in the logfile and for plotting. The need for this comes from the fact that the command the job executes is not descriptive enough.
log_file	Contains the path of the logfile to store everything in.
measuring	Specifies what is to be measured; time, memory or both.

Table 3.2. Specific input settings for logging.These are described in depth in section 3.2.10.

3.2.4 ID - Distinguishing jobs by unique identification

Constructing a test suite is not just a matter of running a single command and then looking at the results. While this might be the case for tiny scripts, testing large systems requires more work. Often, the tester has to set up execution environments first and clean up after. We therefore need a job execution hierarchy where jobs depending on others are executed in a coordinated sequence.

The first step in achieving this hierarchal execution flow is to let each job get a unqiue identifier. This way, they can refer to other jobs they depend on. Non negative integers have been chosen as identifiers.

Explicitly writing a unique id for each job is time consuming. Therefore, our benchmark tool can assign unique ids to jobs automatically. This feature shows the

benefit of using integers over names as identifiers.

3.2.5 Iterations - Executing jobs iteratively

Due to nondeterminism and lack of precision from multiple sources, running a program once is not sufficient to get good results. The tester should therefore specify how many repeated executions are to be done.

If not specified, the default value is to run a command once. If precision is used to dictate when a test has run enough times, the lowest number of iterations is two times. Otherwise it would be impossible to get any information about precision, since we have nothing to compare the results with. If a tester has not specified the number of iterations to use, or explicitly set it to run once, the benchmark tool will automaticallly adjust to run it twice. This is indicated in the results logfile to avoid any confusion.

3.2.6 Cov - Adding precision bound

When measuring execution time, a user might not be satisfied by the precision of the results and decides to raise the number of iterations in hopes of lowering the variance. The better way of doing this is to let the benchmark tool know what the upper bound of desired precision is, and let it run until the results are precise enough.

Unfortunately, achieving the desired precision is not always feasible. A test cannot run forever, so the tester may also set the maximum number of repeated executions allowed.

When using precision as a boundary for termination of a test, the specified number of iterations becomes the maximum number of allowed executions. This is shown in a resulting logfile regardless of what is specified. If the user has not specified any logging and specified a single iteration run, it will run twice without reporting it (because variation can not be calculated from one iteration).

3.2.7 Dependencies - Having jobs depend on other jobs

For jobs that need to wait until other jobs are finished, a list of ids needs to be provided. There might be situations where multiple jobs share the same dependency list, or their separate dependencies are finished executing. In this case, these will be executed in parallel.

3.2.8 Command - Specifying what is to be executed

Setting the command to be run could either be a single bash statement or the execution of a bash script containing multiple statements. Not only is this setting arguably the most important one, but also the only one required by the user. Even though there are quite a lot of usable settings when configuring a job, a user will often only set the command to be executed.

3.2.9 Env - Extending the environment to fit specific tests

For more complex testing, involving multiple execution environment there is a need for environment variables specifying specific paths. Using specified environment variables also provides a way of having specific values for a group of jobs.

These additional environment variables are only visible locally, as each job runs in its own process. Furthermore, there is no way to overwrite the environment variables for any other job (or process).

3.2.10 Logging - Required settings

Providing settings for logging is more involved. All three options are mandatory and therefore described below.

Name - Naming jobs for logging purposes

While an identifier helps us distinguish the jobs in a test, it is not descriptive enough to use as information in a file containing the test results.

One way around this is to use the command to be executed. But the same command can be executed using different environments in an input file. As a result, that is not sufficient. This means that the tester needs to give a name to be used when logging the results.

Log_file - Storing results in a file for later use

In order to write output data to a file, a path must be given. If the given file already exists, it will be overwritten.

Measuring - Deciding what to measure and log

The user must specify what is to be measured. Available options are "time", "memory" or both.

3.3 Remote Execution

In order to be able to run a high number of benchmarks in a reasonable amount of time, a simple way to run benchmarks on several machines is needed. For instance, a set of tests might be divided into several sets where each set is run on a different machine. If such a thing is to be automated, it must include a minimal amount of additional effort for the user or the additional effort might defeat the purpose.

The current solution to remote execution is to keep the entire set of tests on one machine and use scripts to synchronize the working directory to other machines and run them. After evaluating other ways to automate remote execution, this turned out to be the least complicated and error prone way to do it. This chapter describes some methods considered. Section 3.5.3 covers the currently implemented method.

3.3.1 Python libraries for remote objects (Pyro)

Pyro is an extensive library for communication between objects located on different machines through sockets [11]. Once set up, objects can communicate through simple method calls which simplifies programming. However, this requires that all machines involved are set up beforehand and have server or client software running. Therefore, this method was not used.

To run a benchmark remotely via this kind of library, the right python software must be transferred, configured and started on a machine before benchmarking can begin. This requires some extra effort by the user to set up.

Another drawback is that there is no good way to ensure that the programs or scripts used by the tool are available on the remote machine. This could lead to the user creating benchmarks that reference files which will not be available.

3.3.2 Transferring serialized objects

In python, an object can be serialized (called pickled) to text based formats and loaded again [8]. The objects can be loaded on the same machine as well as another one. This can be used to transfer an executor object to a remote machine for loading and execution.

Tests were made where an SSH session was established through which a remote python interpreter was started and objects loaded into that interpreter. Transferring and loading simple objects proved to be simple and effective, giving the ability to access pure python objects on remote machines. This could be done without effort from the user and without requiring more than SSH access and python installed on the remote machine.

Even though simple objects are easy to load, the process becomes more complicated when dealing with non-standard python modules such as third party libraries or user defined classes and modules. In this case, all such modules must be loaded into the remote python interpreter before any instances to any class of those modules are loaded. This could be done either by transferring all files containing any class definition that could be used or trying to transfer serialized versions of all those classes. This could become quite hard to maintain.

Like in the previous section, external software which might be used in the benchmarks might not be available on the remote machine. The user would have to make sure that this is not the case before running the benchmarks.

3.3.3 Transferring the whole working directory

The solution selected and implemented was to transfer the whole working directory to a remote machine and run the program itself remotely. This was done by writing a bash script that uses rsync and ssh to synchronize the working directory, connecting to the remote machine and start the program with the same command line arguments.

A drawback of this method is that all files are transferred to a remote machine, even those that will not be needed. If the working directory contains many files, the file transfer might take some time.

Although simple and somewhat inefficient, this method offers several advantages. First of all, the remote machine will automatically have the same modules in the working directory as the local machine. Secondly, all external scripts are automatically available as well as all software placed in the working directory. This way, the user can place scripts and programs in the working directory and have the benchmarking tool use them regardless of whether the actual benchmark will be performed locally or remotely. The latter is the main reason why this method was selected since this greatly reduces the effort required by the user when testing many different pieces of software. As long as the software resides inside the working directory, it will be copied to all machines that might use it.

3.4 Logging results

Printing out test results on the screen is not enough for longer test suites containing a lot of data. By using a logger module, which isn't too strongly coupled with the rest of the system, logging can easily be extended with functionality for new needs.

3.4.1 Storage

For storing data, CSV format is used. CSV was chosen because of it's simplicity combined with the fact that data from tests are uncomplicated. This works particulary well in python because Python's standard library contains a module for writing and reading CSV.

3.4.2 Using GNUPlot

Developers who need to test different software versions in order to analyze improvement should not have to look at numbers in a CSV formated text file. A plotting script was therefore written, using gnuplot to plot results given by our tool.

Avoiding tight integration into the benchmarking tool

Even though the script is made to handle test results made by the tool exclusively, we decided to implement it as a standalone tool. Extending the tool by the functionallity that the plot script provides, the user would not have to plot the results explicitly. However, doing so would make it less flexible.

Dependencies

Before development of the plotting script began, some research was made to find Python libraries that would make development easier. As expected, a compatible gnuplot library called gnuplot-py was found and examined.

The library turned out to carry a lot of weight in form of dependencies. The author's philosophy has been to keep it as simple as possible, and that means minimizing the number of dependencies required by the tool.

While gnuplot is a given requirement besides Python, we made the decision to skip any gnuplot bindings regardless whether it would have simplified the development or not. As mentioned above, the number of dependencies rose. In addition, this is a plotting script, not a complex program. The extra work was relatively small in comparison to the project as a whole.

Input

The plotting script takes an arbitrary number of files containing results created by the benchmark tool. In each file, each row with the following structure will be considered as something to be plotted:

(measurement that does not begin with #), (result),(standard deviation)

Anything else will be ignored. Information such as number of iterations of execution and date all begin with a # so that those fields are ignored during parsing.

As these files containing data are in CSV format, some simple parsing is done and a new datafile containing gnuplot data is created. However, looking at the format requirement above, GNUplot does not do any calculations regarding precision. The standard deviation is expected with every result.

Output

Output files are created after processing and plotting the input; each containing a single plot. How many files are produced depends on the input. The script will produce a graph for each measurement.

Chapter 4

Examples

To demonstrate the tool and the benchmarks that can be performed with it, benchmarks were created and tested out of three applications from different categories. Firstly, Yaws web server [13] was selected for event driven and network based applications. As a static analysis application with more computing involved, dialyzer [1] was used.

The computer used for producing the example output has the following specifications:

Operating System:	Ubuntu 11.04 (GNU/Linux 2.6.38-13-server x86_64)
CPU:	2x Six-Core AMD Opteron(tm) Processor 2435
CPU cores:	12
CPU clock:	2600 MHz
Memory:	32 GB

The results of this report refer to the tool itself, not the data produced in the examples. This chapter will show a few examples of tests and the kind of output it creates. The data itself can vary depending on compiler settings, test environment and setup of the tests themselves. Therefore, the data presented in this chapter is intended to give a general overview of the capabilities of the tools rather than making conclusions about the software tested.

For that reason, all software was compiled without manually set compiler settings. All applications were installed with all settings set to default.

4.1 Yaws example

4.1.1 About Yaws

Yaws is a web server entirely written in the Erlang programming language and is developed with focus on performance with high concurrency [13]. Yaws uses Erlang as it's main script language for generating dynamic content. According to *Yaws - Yet Another Web Server* [13], this gives high performance during dynamic page generation.

In this example, yaws performance was tested with different Erlang virtual machines.

4.1.2 Test setup

The initialization part is to install a given version on Erlang and then Yaws. Automated scripts were created to perform the installations from arguments passed from the configure file.

When initialization is done, the server is started and also a threaded load generator running simultaneously, making a high number of HTTP requests to the server. When a certain number of requests have been made and answered by the web server, the server and load generator are shut down. While iterating this process, data can be gathered.

The content of the web page being generated and fetched is the Yaws web server home page [13], partly because it is included in the Yaws installation and partly because it is a reasonably large page that also has some Erlang code that is being run to generate the page.

When the measuring is done, the virtual machine as well as the server are deleted so that any subsequent tests are not interfered with.

One of the configuration files used is found in Appendix B. Appendix D contains the script used for installing Yaws. See Appendix C for an example of OTP install script.

The load generator is run on the same machine as the server itself, which might introduce a risk of the load generator itself using too much system resources and affecting the results. To check whether this is the case, the processor load from the server and the load generator were examinied. Figure 4.1 shows the relative processor load from the two processes when running on our test system.

Figure 4.2 shows an example of user time measurement. Figure 4.3 shows an example of system time measurement.

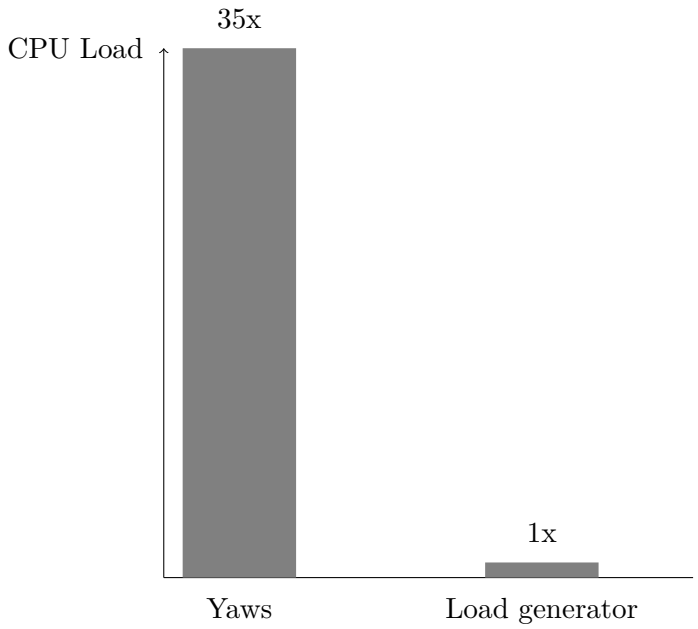

Figure 4.1. Relative CPU load of Yaws web server compared to the load generator on dual Six-Core AMD Opteron(tm) Processor 2435 (2.6 GHz)

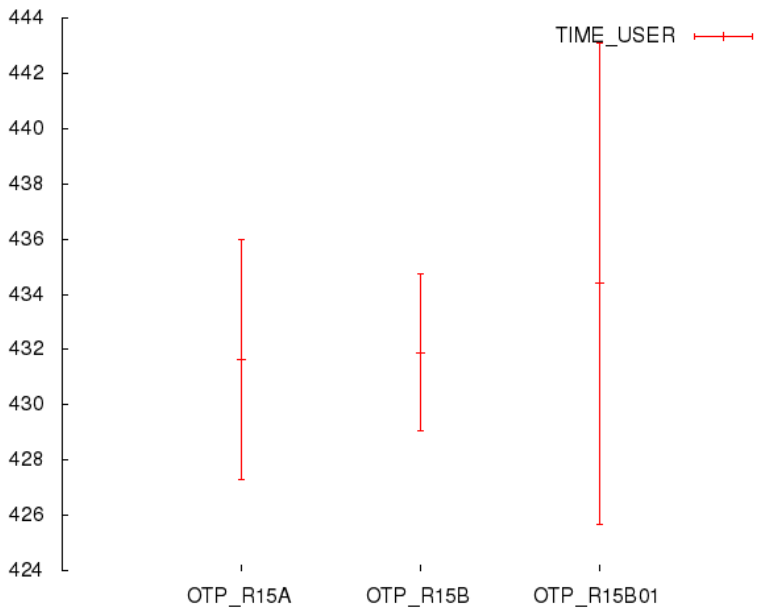

Figure 4.2. Example user time measurement in seconds for Yaws web server with 10000 requests of the Yaws home page index. The bars show the mean average and standard deviation (see Section 2.5).

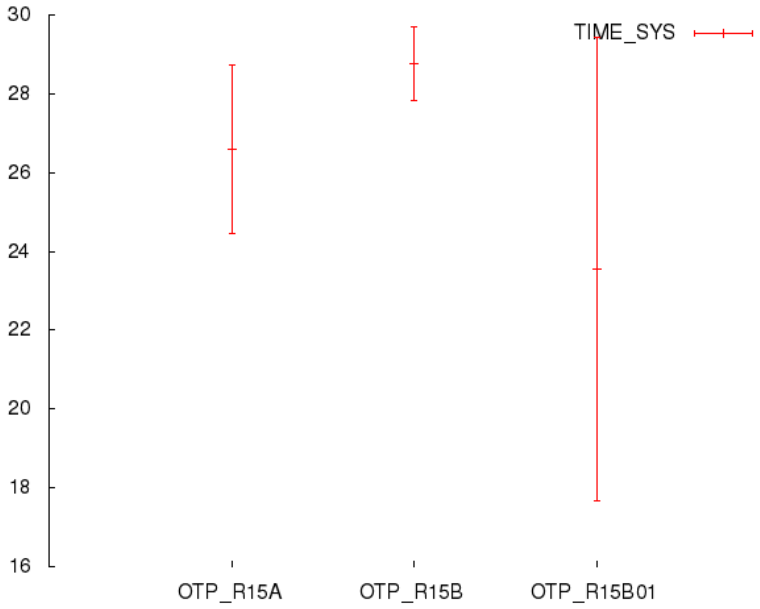

Figure 4.3. Example system time measurement in seconds for Yaws web server with 10000 requests of the Yaws home page index. The bars show the mean average and standard deviation (see Section 2.5).

4.2 Dialyzer example

4.2.1 About Dialyzer

Dialyzer is a static analysis tool for program code in Erlang [1]. The tool analyzes modules or applications to find errors such as type errors, unreachable code and more.

Dialyzer is distributed with the Erlang OTP distribution and installation can easily be done together with Erlang. However, when switching between versions of OTP, the version of Dialyzer might change. Therefore, it is not necessarily safe to compare Dialyzer from one version of OTP to another. The tests in this section can therefore be very inaccurate and should only be seen as an example.

The tool and the example tests are developed with the development of an Erlang JIT compiler in mind. If the JIT compiler is tested in many versions without changing the Dialyzer version, this test would be more relevant.

4.2.2 Test setup

The initialization part consists of downloading and installing OTP. After that, Dialyzer is started from the OTP distribution and set to perform analysis on two OTP applications, namely erts and kernel.

The analysis is iterated and measured and when enough iterations have been run, the OTP installation is removed in the cleanup phase.

One of the configuration files used is found in Appendix A. See Appendix C for an example of OTP install script.

Figure 4.4 shows an example of user time measurement. Figure 4.5 shows an example of system time measurement.

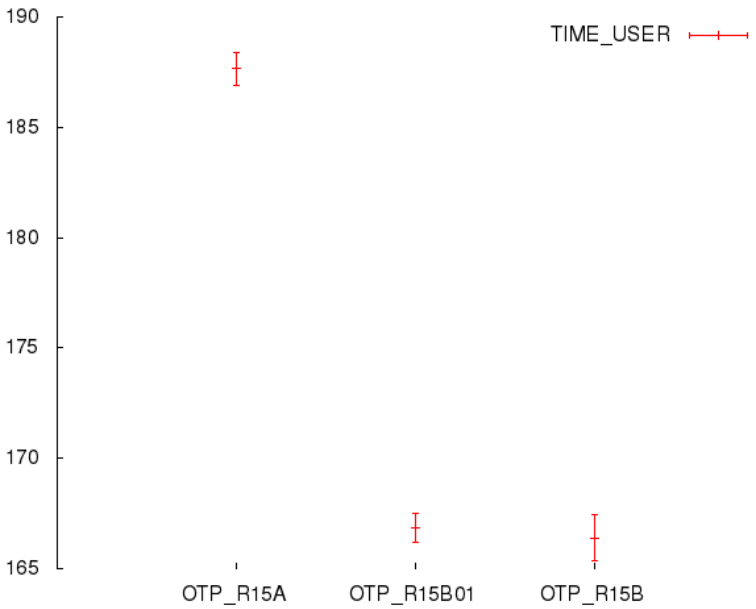

Figure 4.4. Example user time measurement in seconds for dialyzer on OTP erts and kernel. The bars show the mean average and standard deviation (see Section 2.5).

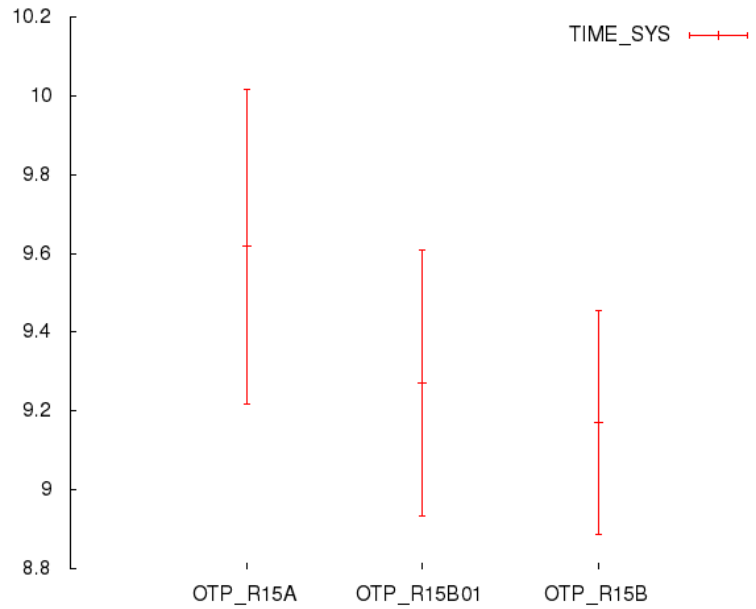

Figure 4.5. Example system time measurement in seconds for dialyzer on OTP erts and kernel. The bars show the mean average and standard deviation (see Section 2.5).

Chapter 5

Analysis

This chapter discusses the example benchmarks as well as the tool in general. Most of the analysis comes from the authors' experiences of the tool while creating and running the example benchmarks. The chapter will mainly present these experiences and pinpoint some strengths and weaknesses of the tool.

5.1 Runscripts

Although not needed for every job, runscripts turned out to be a crucial part of creating a benchmark. In each of our examples, runscripts were used for software installation and in some cases also for running software.

If the runscripts were written flexibly, they could be reused in many configuration files. An example of this is the script that downloads and installs OTP, which is used in all example configuration files.

By moving all application specific functionality to stand-alone scripts, the tools themselves could be kept generic. The tool does not need altering in order to add any benchmark tests.

5.2 Configuration

The configuration files turned out to be easy to work with. All examples have an initialization phase where software is installed, a measurement phase where software is run and measured, and a cleanup phase where the software is uninstalled. Implementing this using inter-job dependencies turned out to work sufficiently well.

Since the configuration files are imported as lists of Python dictionaries, they also have a couple of disadvantages. Below are the two disadvantages found during creation and running of the example configuration files.

5.2.1 Content reuse

The contents of the configuration files could not be reused for several tests. When testing many versions of an application with the same test, each version required its own configuration file with only very slight variations in their content. If there are many configuration files for a benchmark, they could become difficult to maintain since any change would have to be made on many files.

This can, to some extent, be reduced by moving much of the common content to runscripts which can be reused. However, some configuration must stay in the configuration file and will therefore be repeated in each file.

5.2.2 Iterating over multiple jobs

In the tool's current state, there is no support for iterating over more than one job. For example, if starting a server is implemented as a job, creating load as another and killing the server as a third, there is curretly no support for going back to the first job again. Figure 5.1 illustrates such a scenario that currently can not be done using only a configuration file.

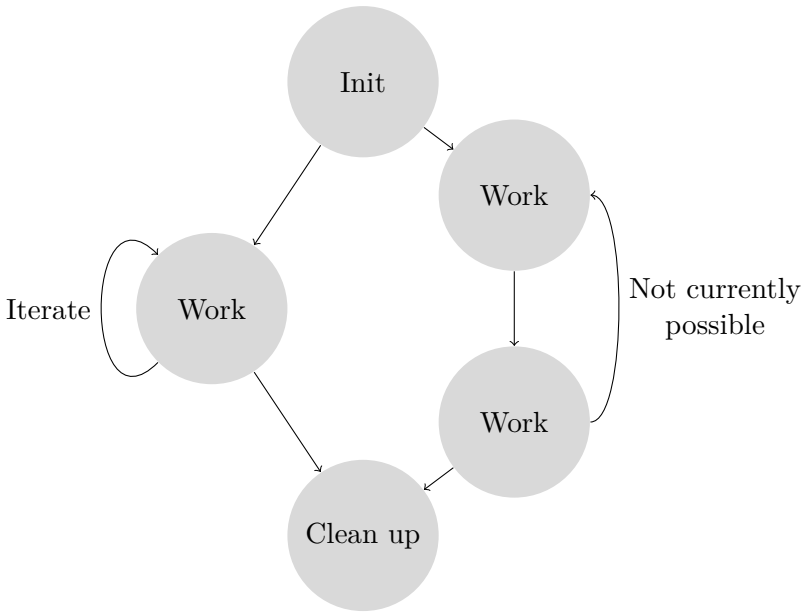

Figure 5.1. Example of configuration that iterates over multiple nodes. This is not currently possible to express using configuration files.

This can be worked around in most cases. In the example test for the Yaws web server, running the server is implemented as one job that is iterated. Both creating load and killing the server is implemented as a single job which is iterated in parallel to the other job. This approach, with two jobs running i parallel and iterated the

same number of times solved the problem for this specific scenario. Although, it requires that there is no dependency between them.

5.3 Resource usage measurement

The resource usage measurement worked well for the example tests. Since all measurement could be done using the system's own resouce usage system call, all this information was easy to obtain.

The drawback of this method was that resource usage could only be measured per process. If more detailed information is needed, for example which parts of the program is using the most resources, a profiling tool would be needed.

5.4 Logging and plotting

The logging and plotting features of the tool proved to be a fast and easy way to get an overall performance measurement of an application. After a test had been run, presenting the result was an easy task.

If the user requires a different output format or plot layout, this would require a bit of extra work by the user. The logging and plotting was intended to provide a quick and simple way of presenting data, which it turned out to be.

Chapter 6

Conclusion

This chapter summarizes conclusions drawn during development and testing of the benchmark tool. Conlusions are based primarily on the analysis chapter.

6.1 Recommendations

This section aims to list the lessons learned when using the tool. In order to use the tool efficiently and easily, the following tips are useful.

6.1.1 Benchmark for comparison - profile for detail

The tool itself does not come with any profiling features. The intention of this tool is to perform comparisons of software in order to give an overview of relative performance.

When optimizing software, overall performance usually gives too little detail about bottlenecks and other performance problems. That is what a profiler is intended for. Therefore, while a benchmark tool can give a good overview of performance increase or decrease, profiling is still needed for the details.

Our recommendation for software optimization is therefore to use the benchmarking tool together with a profiler.

6.1.2 Keep the scripts general

Since content reuse is limited in the configuration files, the scripts and applications called from them should capture as much as possible of the common functionality. This way, one can avoid updating every configuration file if the tests were to change.

6.2 Future work

The time allowed to be spent on the project required that some features were prioritized away. This section describes what was skipped as well as reasons behind the decisions.

6.2.1 Error handling

Error handling is currently done by simple assertions, using python's built in functionality. When an incorrect inputfile is given, assertions will fail and our tool exits unsuccessfully. The problem is that the user will not fully understand the issue from the error messages provided by themselves alone.

We could implement custom made error classes. The benchmark tool is primary constructed for internal testing and these user friendly features have thus not been prioritized.

Efforts have been made to make sure that user input errors are exposed as early as possible.

6.2.2 Iterating over multiple nodes

Flexibility has been the goal in the development of the tool. The current structure allows complex test suites, but there is still room for improvement. One potential flaw is the inability to iterate over multiple nodes. By implementing a new job, this problem can be bypassed. The new job would not execute a command, but contain an executor and run that multiple times. This hierachial functionality allows more complex benchmarks.

Bibliography

[1] Ericsson AB. *Dialyzer manual*. 2012. URL: http://www.erlang.org/doc/man/dialyzer.html (visited on 05/10/2012).

[2] Ericsson AB. *Erlang FAQ. 1. What is Erlang*. 2012. URL: http://www.erlang.org/faq/introduction.html (visited on 05/18/2012).

[3] Ss Um An and Guido Van Rossum. "Python for Unix/C Programmers Copyright 1993 Guido van Rossum 1". In: *Proc. of the NLUUG najaarsconferentie. Dutch UNIX users group*. 1993.

[4] Mikael Östberg. "UTS: A Portable Benchmark for Erlang/OTP". Bachelor Thesis. Swedish Institute of Computer Science (SICS), The Royal Institute of Technology (KTH), 2010.

[5] Python Software Foundation. *Python v2.7.3 documentation. 35.13. resource - Resource usage information*. 2012. URL: http://docs.python.org/library/resource.html (visited on 04/22/2012).

[6] Python Software Foundation. *Python v2.7.3 documentation. 17.4. signal — Set handlers for asynchronous events*. 2012. URL: http://docs.python.org/library/signal.html (visited on 06/07/2012).

[7] Python Software Foundation. *Python v2.7.3 documentation. 15.1 os - Miscellaneous operating system interfaces*. 2012. URL: http://docs.python.org/library/os.html (visited on 05/18/2012).

[8] Python Software Foundation. *Python v2.7.3 documentation. 11.1 pickle - Python object serialization*. 2012. URL: http://docs.python.org/library/pickle.html (visited on 05/21/2012).

[9] David Goodger. In: (July 2011). URL: http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html (visited on 05/18/2012).

[10] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6. getrusage*. 2004. URL: http://pubs.opengroup.org/onlinepubs/009604599/functions/getrusage.html (visited on 04/22/2012).

[11] Irmen de Jong. *Pyro Manual*. 2012. URL: http://packages.python.org/Pyro/ (visited on 05/21/2012).

[12] Guido van Rossum. *A Brief Timeline of Python*. 2009. URL: `http://python-history.blogspot.se/2009/01/brief-timeline-of-python.html` (visited on 05/18/2012).

[13] Claes Wikstrom. *Yaws - Yet Another Web Server*. 2011. URL: `http://yaws.hyber.org/yaws.pdf` (visited on 05/10/2012).

Appendix A

Dialyzer configuration example

```
1  [
2    { "command"    : "scripts/build_OTP_from_git.sh https://github.com/erlang/otp.git "
3                   + "--working_dir /tmp/x-bench --checkout OTP_R15B01 --disable-hipe" },
4
5    {
6      "command"    : "/tmp/x-bench/otp/bin/dialyzer --build_plt --apps erts kernel",
7      "logging"    : {"measuring" : ["time"],
8        "name"     : "dialyzer-R15B01-erts-kernel",
9        "log_file" : "dialyzer_R15B01"},
10     "iterations" : 10
11   },
12
13   { "command"    : "rm -rf /tmp/x-bench" }
14 ]
```

Appendix B

Yaws configuration example

```
[
  {
    "id"           : 1,
    "command"      : "scripts/build_OTP_from_git.sh https://github.com/erlang/otp.git "
                   + "--working_dir /tmp/x-bench --checkout OTP_R15B01",
    "dependencies" : []
  },
  {
    "id"                : 2,
    "command" : "scripts/yaws/build_yaws_from_git.sh https://github.com/klacke/yaws.git "
     + "/tmp/x-bench/otp scripts/yaws/yaws.conf --working_dir /tmp/x-bench --disable-pam",
    "dependencies"      : [1]
  },
  {
    "id"            : 3,
    "command"       : "scripts/yaws/run_yaws.sh /tmp/x-bench/yaws",
    "dependencies"  : [2],
    "logging"       : {"measuring": ["time", "memory"],
      "name": "OTP_R15B01",
      "log_file": "log_yaws_R15B01"},
    "iterations"    : 100
  },
  {
    "id"      : 4,
    "command" : "python scripts/yaws/run_load_generator.py 12 10000 http://127.0.0.1:8000 && "
              + "scripts/yaws/stop_yaws.sh /tmp/x-bench/yaws",
    "dependencies" : [2],
    "iterations"   : 100
  },
  {
    "id"           : 5,
    "command"      : "rm -rf /tmp/x-bench",
    "dependencies" : [3, 4]
  }
]
```

Appendix C

OTP installation script example

```
1  #!/bin/bash
2
3  # Script for fetching and building OTP from git repository.
4
5  function print_help_and_exit {
6      echo 'usage: ./build_OTP_from_git.sh <URL> [--checkout <commit/branch/tag>]\
7   [--working_dir <dir>] [<config_arg> ...]'
8      exit
9  }
10
11 if [ $# -lt 1 ]
12 then
13     print_help_and_exit
14 fi
15
16 # First parameter must be URL of git repo:
17 URL=$1
18 WORKING_DIR=.
19 CHECKOUT=""
20
21 shift
22
23 while [ "$1" != "" ]
24 do
25     case $1 in
26         --checkout )
27             CHECKOUT=$2
28             ;;
29         --working_dir )
30             WORKING_DIR=$2
31             ;;
32         * )
33             break
34     esac
35     shift
36     shift
37 done
38
39 mkdir -p $WORKING_DIR
40 cd $WORKING_DIR
41 git clone $URL otp
42 cd otp
43
```

```
44  if [ "$CHECKOUT" != "" ]
45  then
46      git checkout $CHECKOUT
47  fi
48
49  ./otp_build autoconf
50  ./otp_build configure --prefix=$(pwd) $*
51  ./otp_build boot
52  make install
```

Appendix D

Yaws installation script example

```
1  #!/bin/bash
2
3  # Script for fetching and building yaws from git repository.
4
5  function print_help_and_exit {
6      echo 'usage: ./build_yaws_git.sh <URL> <OTP_ROOT> <CONF_TEMPLATE>\
7   [--checkout <commit/branch/tag>]\
8   [--working_dir <dir>] [<config_arg> ...]'
9      exit
10 }
11
12 if [ $# -lt 3 ]
13 then
14     print_help_and_exit
15 fi
16
17 # First parameter must be URL of git repo:
18 URL=$1
19 OTP=$2
20 CONF_TEMPLATE=$3
21 WORKING_DIR=.
22 CHECKOUT=""
23
24 shift
25 shift
26 shift
27
28 while [ "$1" != "" ]
29 do
30     case $1 in
31         --checkout )
32             CHECKOUT=$2
33             ;;
34         --working_dir )
35             WORKING_DIR=$2
36             ;;
37         * )
38             break
39     esac
40     shift
41     shift
42 done
43
```

```
44 OLD_DIR=$(pwd)
45 mkdir -p $WORKING_DIR
46 cd $WORKING_DIR
47 git clone $URL yaws
48 cd yaws
49
50 if [ "$CHECKOUT" != "" ]
51 then
52     git checkout $CHECKOUT
53 fi
54
55 export ERL=$OTP/bin/erl
56 export ERLC=$OTP/bin/erlc
57 autoconf
58 ./configure --prefix=$(pwd) $*
59 make
60 make install
61
62 cd $OLD_DIR
63 echo $(pwd)
64 sed s/'<YAWS_DIR>'/$(echo $WORKING_DIR/yaws \
65 | sed s/'\/'/'\\\/'/g)/g $CONF_TEMPLATE > $WORKING_DIR/yaws/etc/yaws/yaws.conf
```