A Virtual Network Function Workload Simulator

Mikael Mollberg
Abstract

A Virtual Network Function Workload Simulator

Mikael Mollberg

The purpose of this thesis was to design and implement a program that can simulate configurable workload similar to a Virtual Network Function (VNF). When designing a Network Function Virtualization solution the contents of the environment are not always known and in these cases it would be helpful to have a simulator that can generate workload similar to VNFs. Characteristics of applications were studied and the simulator presented in this paper is able to generate workload based on a given configuration.

Handledare: Viktor Leijon
Ämnesgranskare: Lars-Ake Nordén
Examinator: Olle Gällmo
IT 16 073
Tryckt av: Reprocentralen ITC
Contents

1 Introduction 1

2 Design 2
  2.1 Time span ................................. 2
  2.2 Workload .................................. 3
    2.2.1 CPU .................................. 4
    2.2.2 Network ................................ 4
    2.2.3 I/O .................................. 5
    2.2.4 Memory ................................. 5
  2.3 YANG Model ............................... 5

3 Implementation 7
  3.1 Main .................................... 8
  3.2 Task handler .............................. 8
  3.3 Work ................................... 8
  3.4 Timing ................................... 9
  3.5 Random Generation ....................... 10
  3.6 Time span ................................ 10
  3.7 CPU ................................... 10
  3.8 Network .................................. 11
  3.9 I/O ..................................... 11
  3.10 Memory ................................. 12

4 Results 12
  4.1 Time spans ................................ 12
  4.2 CPU ..................................... 14
  4.3 Network .................................. 16
  4.4 I/O ..................................... 17
  4.5 Memory ................................. 18

5 Related Work 19

6 Conclusions & Future Work 19
1 Introduction

Configuring a large network infrastructure has always been hard [1]. It can contain numerous components that need to be managed by network operators. In addition, all unforeseen events must be handled by the operator and configurations must be adapted to the continuously changing state of the network. Software defined networking (SDN) allows the operator to change configurations from a controller instead of changing it manually on each individual component in the network.

Adding new services to a network is costly [2], each component takes storage, energy and must be affiliated with an operator that can maintain them. New services may require additional components to be installed. Network Function Virtualization (NFV) was introduced as a way to ease these problems, instead of installing new hardware the services can be virtualized in another machine. Moving the services into the cloud simplifies deployment, scaling and management of network services. The need for hardware specific machines is eliminated when the services can be virtualized on a general purpose machine.

![Network Function Virtualization](image)

When moving network functions into the cloud the Virtual Network Function (VNF) plays a crucial role. However, in many generic situations such as orchestration, performance testing and provisioning of cloud solutions it is important that the solution can work for many different VNFs, with many different loads. For this reason it would often be helpful to have a simulator that can have many of the load characteristics of a real VNF, and that can be configured to simulate the behavior of many different VNFs.

The purpose of this thesis is to design and implement a simulator that can generate workload based on a given configuration. Configuring the simulator is to be done with ConfD[3], a management agent framework for configuring network devices. This paper presents the design and implementation of the simulator and includes results from tests done on the program. The contributions are as follows:

- A detailed description of the design of the simulator and the configuration
model is provided.

- The implementation of the simulator is described.
- Multiple tests have been made and the results of the simulator are shown.

2 Design

Generating generic workload requires a well designed simulator. The configuration must be simple and still have enough parameters for a user to specify a workload pattern. Having too few options will make the simulator trivial but having too many reduces the usability and complicates it.

The work categories that must be taken into consideration are the CPU usage, network traffic, memory access and disk input/output. It must be possible to configure the simulator so that it behaves similarly to a defined VNF for each category.

There are two common approaches used when it comes to generating synthetic workload [4]. The empirical approach uses traces of an existing application and replays the sampled workload. In this case, where the type of VNF is unknown, the second, analytical, approach is more appropriate. Instead of sampling the workload of a program it uses mathematical models and the program characteristics to generate the desirable workload.

Two important aspects when studying the workload of a program is the amount of work that is done and the time it takes for the work to finish. If a program is computation heavy, then how the CPU usage changes over time is what defines the workload, as well as how long it is computing. For each category of work there must be options about how much workload that should be generated as well as the duration of the work.

By defining different patterns that decide how much work is to be done, depending on the configuration, the user can specify workload that is similar to a desired VNF. The patterns used for the simulator are static, normal distribution, periodical, interval and linear. These five different patterns allow configuration following simple mathematical functions.

All workload generation is divided into tasks. Each task makes sure that the right amount of work is generated during the configured time.

2.1 Time span

Instead of specifying the time for the workload the configuration of time is separated into an additional entity. The time spans are associated with some work and allow configuration of when the work is going to be running and sleeping. There are five different patterns that can be used for each time span. Each pattern characterizes the time of execution differently.

**Static** Constant execution.
Parameters: None

**Normal** Values are normally distributed.
Parameters: run expected, run deviation, sleep expected, sleep deviation.
Periodical Single values.
Parameters: run, sleep.

Interval Values are uniformly distributed in an interval.
Parameters: run high, run low, sleep high, run low.

Linear Values are increasing/decreasing with a start and stop value and a number of steps. It restarts after all steps have been taken.
Parameters: amount of steps, direction, run start, run stop, sleep start, sleep stop.

There are two time spans to enable more specific execution time configurations. The first time span is the period, it is intended to be longer than the second time span, the cycle. The work that the period is associated with is the cycle. This means that the period decides when the cycle is going to run and when not to. The cycle is then associated with some workload and the running time is intended to be shorter than the period. Together they can create a wide variety of different workload patterns. An example period and cycle configuration could be as follows:

![Time spans](image)

**Period:** interval
- Run high: 10s
- Run low: 7s
- Sleep high: 2s
- Sleep low: 0.5s

**Cycle:** normal
- Run expected: 0.5s
- Run deviation: 0.1s
- Sleep expected: 0.1s
- Sleep deviation: 0.1s

Figure 2: Plot of the example configuration for some undefined work. The small gaps at the bottom is the cycle sleeping time. The large gap is the period sleeping time.

### 2.2 Workload

Similarly to time spans, every work category has a type of pattern. The pattern decides which configuration parameters are available and how the generated work changes. What the value is depends on the type of work.

**Static** The parameters are constant.
Parameters: single value that does not change.
**Interval** The parameters are uniformly distributed in an interval. Parameters: value low, value high.

**Normal** The parameters are normally distributed. Parameters: value expected, value deviation.

**Linear** The parameters increase/decrease linearly. Parameters: amount of steps, direction, value start, value stop.

### 2.2.1 CPU

The usage of a CPU can be measured in time or percentage. When the CPU does some instruction it is always using its full capacity. This means that having a CPU usage of 12% does not say anything unless the total time is known. Generally the usage is per second, but the CPU does not run for 0.12 seconds and then sleep for the 0.88 seconds remaining. The instructions are distributed over time and are executed when needed. To generate an average load the simulator keeps track of how much time each calculation takes. By using that time it can calculate how much sleep time is needed to get a certain CPU usage.

\[
sleeptime = \frac{\text{runtime}}{1 - \text{percentage}} - \text{runtime}
\]

The simulator uses the source code from stress-ng [5] to generate CPU workload. Stress-ng is an open source program for stress testing a computer system and has over 50 different CPU methods. Instead of maximizing the CPU usage this modified version only runs when the cycle is running and generates load according to the configuration.

### 2.2.2 Network

Traffic generation between two components in a network is fairly easy to accomplish. A common practice is to use the client-server model and allow multiple clients to communicate through the server. In this case the simulator will act as the (possibly only) client. Because the amount of data that the simulator wants to send can vary there must be a protocol established in advance so that both the client and the server knows what to expect. A simple solution is to first send how many bytes the data is and then send the actual data. This allows the server to prepare itself before receiving the data. Sending more data than a server is expecting will cause an overflow and the exceeding data will be lost. The simulator should both send and receive data. By using a server that acts as a mirror the simulator will first send data and then receive it.

---

**CPU Methods**

Some of the CPU calculations done are Ackermann, Floyd–Steinberg dithering, Euler, Fibonacci, Fast Fourier Transform, Fowler–Noll–Vo hash, Hamming, Hanoi, Jenkin’s, Queens problem and many double, float and integer operations. See the stress-ng manual for a complete list.
To send data the simulator needs to known the address and port of the target server it is connecting to. When the simulator has connected it can start to send the data.

2.2.3 I/O

Disk usage is generated by reading and writing to a file. The parameters are the total size of the file and the read/write distribution in percentage where 0 is read-only and 100 is write-only. The last parameter is the read/write size in bytes. To be able to read from the file there data must already exist in it. The file is therefore filled with data when the work starts. All data read and written from the file starts at a uniformly distributed position and reads/writes the amount of bytes specified in the configuration. A temporary storage is used to read to or write from. It is discarded after each time.

2.2.4 Memory

Workload generation for memory is done by reading and writing to the systems virtual memory. The parameters are the total amount of bytes to be allocated and the amount of bytes that should be read/written. Writing data is done by filling the memory with randomly generated values a byte at a time. The positions that data are written to are uniformly distributed over the allocated memory. Reading is done in the same way except that no data is changed, only read.

2.3 YANG Model

YANG [6] is the data modeling language used for defining the simulator configuration. ConfD uses the model when generating the configuration database and interfaces. The model is a very important part of the simulator because all other components depend on it. Changing the model requires changes in the program and this is something that should be avoided.

YANG models are defined by containers, leaves, groupings and type definitions. A container can consist of multiple leaves and use different groupings. A grouping is content that may be used in multiple containers. A leaf is the actual configuration data. Each leaf has a type that then decides what values it can have. Custom types can be defined by using a type definition.
typedef percentage {
  type uint8 {
    range "0 .. 100";
  }
}

The model is designed with the same idea of dividing work into tasks but here all workload is separated into lists instead. This increases the visibility for the user when adding work as it shows what type of work exists and what is being added. Every list is associated with a work category and the grouping time_span and can have any positive number of entries. The time_span grouping contains the period and the cycle. These two have different documentation and therefore uses two different pattern groupings, but the configuration used in both is the same. All work containers have their own leaves for parameter configuration but the patterns that can be used are gained from a grouping. See figure 3.
3 Implementation

The implementation of the simulator is divided into several different modules. The core of the program is the main module, it connects all the pieces and takes care of the communication with ConD. It reads from the ConD database and creates tasks for each defined work. After creating the tasks they are given to the task handler. It makes sure that each task starts running when the simulator
is started as well as destroying all tasks when it is stopping. Each work category
has its own module, but the work module acts as a container for every work. It
holds information about which functions to run and handles the run conditions
of each work. Because the work module is so generic and the time spans are
also considered as work and are run as tasks by the task handler.

3.1 Main

Before starting, the simulator must open a socket and connect to ConfD. When
the connection is established a subscription is created and the current config-
uration is read. To get updates the simulator continuously polls ConfD for
changes. For every time a change occurs the simulator discards the previous
configuration and reads from the database again.

All configurations are stored in ConfDs built-in XML database. Reading
from the database is done using the socket together with the XML tree path
in the form of a string. Reading the run condition of the simulator is done by
using the socket, a variable and the XML tree path:

```c
int run;
cdb_get_bool(rsock, &run, "/simulator/run");
```

Because all tasks are stored in lists the data is read by first getting the amount
of entries and the list, and then iterating over them indexed from 0.

```c
n = cdb_num_instances(rsock, "/simulator/tasks/cpu");
for (int i = 0; i < n; i++) {
  ...
  cdb_get_enum_value(rsock, &work_type,
                     "simulator/tasks/cpu[%d]/work/pattern", i);
  ...
}
```

Here %d is replaced with the current iteration number.

A task is created for each entry in the work lists and data is read for the pe-
riod, cycle and workload parameters. When all data has been read the simulator
starts all the tasks.

3.2 Task handler

The tasks are taken care of by the task handler. It keeps a list of all the tasks
and is started after the configuration is read from ConfD for the first time. The
list is implemented with the BSD queue library [7]. When the handler starts
it iterates over all tasks in the list and starts each task in a separate thread.
The handler then waits until the simulator either stops or gets an update from
ConfD. Each second it checks if the stop variable is set or not. If it is not set it
sleeps for another second and repeats this until the simulator tells it to stop.

3.3 Work

The work module is a generic container for any type of work. It has a data
structure for storing which functions to use for running and destroying the
work, thread id and mutexes for handling the thread as well as run condition variables.

```c
struct work {
    void *data;
    void (*)(run_f)(void *);
    void (*)(destroy_f)(void *);
    pthread_t thread;
    pthread_mutex_t run_lock;
    pthread_mutex_t stop_lock;
    int run;
    int stop;
};
```

When creating the work there are several parameters that must be supplied. The data is a pointer to the information needed by the work. The second and third parameters are the two function pointers. The first is used for generating the workload and the second for destroying the work when it is stopped.

Every run function must contain a call to the function `work_stop_check`. This function checks if the stop variable has been changed and will return 1 if the work should stop and 0 if it should continue. It also checks the run variable and if it has changed to 0 it will enter a loop. Until the run variables has been reverted to 1 the work thread stay inside the loop, stopping any work from being executed. This is used when one of the time spans temporarily tells a work to stop running.

Because the work data structure contains all locks it must be passed to the run function and must be the only parameter used. When the run function is called each work category then gets the data needed from the data pointer. This is done by type casting it to the struct or variable that is used. The basic structure of each run function is as follows:

```c
void some_work_run(Work work)
{
    Some_work some_work = work_data_get(work);
    ...
    while(!work_stop_check(work)) {
        ...
    }
}
```

When the handler tells the work to stop the stop variable will be changed to 1. The handler will then wait until the work thread has reached its end and then runs the destroy function and frees all allocated data.

### 3.4 Timing

There are many time calculations in the simulator and it is important that these are as accurate as possible. The library used is `time.h` that is a part of the GNU C library and it provides functions for calculating time in nanoseconds. To get the current time the function `clock_gettime` is used.
The time is stored in a `timespec` struct that stores time in seconds and nanoseconds. Because the `timespec` struct stores seconds and nanoseconds separately the value for nanoseconds will never go above a billion, which is the same as a second. When `clock_gettime` is called the second time the two results can be compared. If the nanoseconds in the second result is less than in the first result then the duration for the nanoseconds must be recalculated and the duration in seconds decreased by one.

```c
if (start.tv_nsec < end.tv_nsec) {
    result.tv_nsec = BILLION - start.tv_nsec + end.tv_nsec;
    result.tv_nsec = start.tv_nsec - end.tv_nsec - 1;
} else {
    result.tv_nsec = end.tv_nsec - start.tv_nsec;
    result.tv_nsec = end.tv_nsec - start.tv_nsec;
}
```

### 3.5 Random Generation

Random numbers are used in several modules of the simulator. As pointed out by G. Marsaglia [8] many common random number generators have defects that are unsuitable for simulations. To assure that the numbers generated are statistically good the simulator uses PCG [9].

### 3.6 Time span

Creating a time span requires a work and a pattern. The work is the workload that is going to be running when the time span is running. The time span runs depending on which pattern is used and the parameters that have been read from the CondD database.

Regardless of which patterns is used each time span behaves very similarly. The exception is the `static` pattern where the work is always running. Each time span has a time where the work is going to be running and one where it is going to be sleeping. Because the length of the times are known the time span can simply sleep during that period.

```c
while (!work_stop_check(work)) {
    work_run(work);
    sleep_time_sleep, rem_sleep);
    work_stop(work);
    sleep_time_sleep, rem_sleep);
}
```

### 3.7 CPU

For this workload the stress-ng source code stands for most of the implementation. There must still be a task for it so that the task handler can tell the work to run and stop. When `cpu_work_run` is executed it first checks what intensity has been chosen. If the pattern is static and the intensity is at 100 then it will go into a loop and generate workload constantly. For the other patterns it must calculate how much time it needs to sleep to generate appropriate load. This is done by starting the clock before executing a function and stop directly after.
The functions that are called comes from stress-ng. It uses a list to store all functions and walks through the list sequentially, incrementing the counter each time a function is called.

3.8 Network

Because all data may not be transferred immediately, the socket must be read and written until all data has been handled. All the data is associated with a buffer. A counter is used to keep track of how much data has been handled. The function call to the socket will return how much data has been processed and by using that the data can continue to be written or read.

```c
int write_data(int sock, int len, void *buffer)
{
    int bytes_written = 0, n = 0;
    while (bytes_written < len) {
        n = write(sock, buffer + bytes_written,
                   len - bytes_written);
        if (n < 1)
            return -1;
        bytes_written += n;
    }
    return 0;
}

int read_data(int sock, int len, void *buffer)
{
    int bytes_read = 0, n = 0;
    while (bytes_read < len) {
        n = read(sock, buffer + bytes_read, len - bytes_read);
        if (n < 1)
            return -1;
        bytes_read += n;
    }
    return 0;
}
```

3.9 I/O

A file is opened with the function fopen and the option w+ for writing and reading as well as creating a file if it does not exist. When writing and reading to the file a char buffer is used for storing temporary data. The position is generated with PCG and the file pointer is then set to that position.

```c
pos = pcg64_boundedrand(tot - len);
fseek(fp, pos*sizeof(str), SEEK_SET);
```

A read or write is then done with the set file pointer.
fwrite(str, sizeof(str), 1, fp);
fwrite(str, sizeof(str), 1, fp);

3.10 Memory

The memory is allocated with `malloc` and the address stored to a pointer. When calling the functions that reads/writes the pointer to the allocated memory and the length of the read/write are included as arguments. The pointer is type casted to a `uint8_t` pointer to make it one byte long. The start location is a number between 0 and `total memory size – length` that is randomly generated with PCG.

```c
start = pcg64_boundedrand(tot – len)
end = start + len;
```

The pointer is then moved to the start position and memory will be written/read until the end is reached. Writing is done by setting the value at the memory location the pointer is pointing to. The value is a randomly generated number between 0-255 (255 is the maximum 8-bit value). Reading is

```c
Write
while (start++ < end) {
  x = pcg64_boundedrand(256);
  *(void*)((ptr++) = x);
}

Read
while (start++ < end) {
  *(void*)((ptr++) = x);
}
```

4 Results

In this section the results from tests with the simulator will be shown. All tests were done on a Dell XPS 9550 running Ubuntu 16.04 LTS. It has a sixth generation Intel Core i7-6700HQ with a clock rate up to 3.5GHz (2.6GHz as standard), four cores and 6MB cache, 16GB DDR4 memory with 2133 MHz on two channels (8GB x 2) and a Samsung PM951 NVMe SSD with 512GB of storage.

All configuration of the simulator was done using ConfD and the ConfD command line interface.

The data seen in the results was retrieved with different Linux command line tools. `Top` was used to get the CPU usage of the simulator, `dstat` the network traffic generated and `sysdig` for disk usage.

4.1 Time spans

For testing the time spans static CPU workload is generated with 100 in intensity. The expected result would be that the work is at 100 when the time spans are running.

In figure 4 the cycle is using the linearly increasing pattern and the period the interval pattern. The sleep time of the cycle can be seen in the smaller dips. The first dip has a lower time than the last dip, showing that the cycle sleeping time is increasing. The same applies to the run times, as the first running times increase over time. The sleep times for the period is seen in the larger
gaps. The difference between the gaps is small but the first gap is a bit longer than the second.

**Period**: interval
- Run low: 8s
- Run high: 12s
- Sleep low: 3s
- Sleep high: 5s

**Cycle**: linear
- Direction: increasing
- Steps: 10
- Run start: 2
- Run stop: 4
- Sleep start: 0.1
- Sleep stop: 2

Figure 4: Configuration and graph of the workload when using time spans with the interval and linear patterns.

The second test uses the normal and periodical patterns. The times are a bit shorter than if the previous test and can be seen in figure 5. The cycle times are the repeating short tops and bottoms. The period sleeping times are the longer gaps with varying duration in the graph. An additional gap should be between 5 and 18 seconds as the running time is expected to have a value of 5 seconds.
Period: normal
Run expected: 5s
Run deviation: 1s
Sleep expected: 2s
Sleep deviation: 0.5s
Cycle: periodical
Run: 1s
Sleep 0.3s

Figure 5: Configuration and graph of the workload when using time spans with the normal and periodical patterns.

4.2 CPU

The tests for CPU workload check the CPU usage the simulator process uses. The update interval between each check is between 100 and 200 milliseconds. In the figures below the dashed line is for the average intensity over the whole time period.

In figure 6 the intensity is uniformly distributed between 20 and 40. The patterns used for the period and cycle are periodical and interval. There are multiple occasions where the intensity is much higher than 40 or much lower than 20. The average intensity is approximately 23 and is reasonable considering the sleeping times.
**Period**: periodical  
Sleep: 2.0s  
Run: 4.0s  
**Cycle**: normal  
Sleep expected: 0.2s  
Sleep deviation: 0.1s  
Run expected: 1.0s  
Run deviation: 0.5s  
**Work**: interval  
Intensity low: 20  
Intensity high: 40

![CPU Workload](image)

Figure 6: Configuration and graph of CPU workload with the interval pattern.

The second test uses a normally distributed intensity with a mean of 70 and 10 in standard deviation. The highest value seen in figure 7 is a bit over 80 and the lowest a bit below 50, possibly lower although it could be affected by the the time spans sleeping time. The average is around 46.

**Period**: periodical  
Sleep: 2.0s  
Run: 4.0s  
**Cycle**: linear  
Direction: increasing  
Amount of steps: 10  
Sleep start: 0.1s  
Sleep stop: 1.0s  
Run start: 0.1s  
Run stop: 3.0s  
**Work**: normal  
Intensity expected: 70  
Intensity deviation: 10

![CPU Workload](image)

Figure 7: Configuration and graph of CPU workload with the normal pattern.
4.3 Network

The network tests were done on a 100/100 Mbps internet connection to a server outside the local network with a connection of 100/10 Mbps. The server was running on a Lenovo Thinkpad X201 with a 2.4Ghz Intel Core i5 and 4GB RAM.

In figure 8 all patterns are static and the bytes sent were 10000. The graph shows the maximum throughput that is achievable on this setup. The sudden decrease in data in the middle is possibly because of a timeout.

![Network Workload](image)

**Period**: static  
**Cycle**: static  
**Work**: static  
**Bytes**: 10000

Figure 8: Configuration and graph of network workload with the static pattern.

The second test uses a periodical cycle which creates a graph with multiple spikes.
Period: static  
Cycle: periodical  
Sleep: 3.0s  
Run: 2.0s  
Work: static  
Bytes: 10000

Figure 9: Configuration and graph of network workload with static work and a periodical cycle.

4.4 I/O

The following graphs show how much data has been written and read to the disk with an update frequency of one second. The blue lines are data read and green data written.

The first test was done using a static period and a periodical cycle. The read/write distribution was 40 meaning that reads were more likely to occur than writes. In figure 10 the data read and written are shown over a period of 30 seconds.
**Period:** static  
**Cycle:** periodical  
Run: 0.5s  
Sleep: 0.1s  
**Work:** static  
Read write distribution: 40  
Bytes: 2000

Figure 10: Configuration and graph of i/o workload.

The second test was done using the normal pattern for the period, the interval pattern for the cycle and the linear pattern for the work. There is more data written than read which can be seen in figure 11.

**Period:** normal  
Run expected: 1s  
Run deviation: 0.05s  
Sleep expected: 0.1s  
Sleep deviation: 0.05s  
**Cycle:** interval  
Run low: 0.1s  
Run high: 0.3s  
Sleep low: 0.01s  
Sleep high: 0.1s  
**Work:** linear  
Unique bytes: 10000000000  
Read write distribution: 75  
Direction: increasing  
Steps: 20  
Bytes start: 7000  
Bytes stop: 2000

Figure 11: Configuration and graph of i/o workload.

### 4.5 Memory

No tool for monitoring the memory usage of a process that gives information about the amount of data read and written to memory has been found. There
are tools that show the total amount of memory used by a process, but after
the allocated memory has been filled there is no nothing that says how much
data is read and written.

Using a configuration with a static period, a periodical cycle with 1.2 in run
and 0.4 in sleep and a normally distributed work with 3000000000 in unique
bytes, 20000 expected bytes and 1000 in deviation for the test. This test was
monitored using top. When starting the simulator the resident memory of the
process is increased from 1672 bytes to 2073MB during the first cycle, it then
continues to increase to 2709MB and 2859MB in the following two cycles until
it stops at 2861MB. Data is still written but it is not shown how much.

5 Related Work

Prior work in this area is limited. There are tools that simulate cloud infra-
structures and the workload of cloud computing applications as well as tools for
generating high load to systems but not that allow the kind of configuration
that can simulate the workload characteristics of a VNF.

CloudSim [10] is a framework for simulating cloud computing infrastructures
and management services.

Rain [11] and Cloudstone [12] are workload generators that simulate web 2.0
applications.

VMmark [13] is a benchmark for measuring the virtualization performance
of systems and deliver reliable values for system comparison.

Stresslinux [14] is a minimal Linux distribution for generating high load and
stressing a system.

6 Conclusions & Future Work

The simulator presented in this paper provides a way to simplify testing of vir-
tualization solutions by applying load to the system. It is configurable and can
generate CPU, network, memory and I/O workload based on the configuration.
The design focuses on being simple but still allows the user to specify workload
with many different characteristics. There are, however, many different ways to
design the simulator.

For future work, it would be of interest to study how the simulator behaves
in a virtual environment as well as what modifications can be done to increase
the functionality.

Find additional ways to implement the workload generation and increase the
number of actions applied for each work category.

Monitor VNFs and study traces to get more information about the applica-
tions characteristics and how it compares to the simulator.
References


