**Input Sensitive Profiling for Tasteful Server**

**CS 474: Object Oriented Languages**

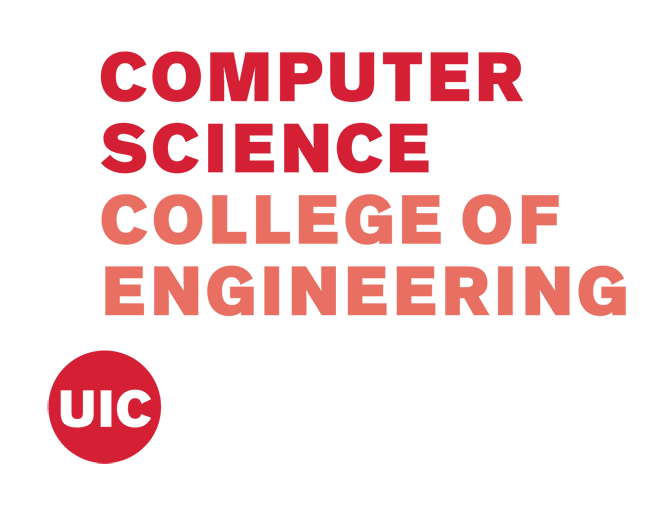
**And Environments**

**Course Project Report**

**Abhijay Patne**

**UIN: 663324999**





**Table of Pointers**

[**Application Details**](#appdetails)

[**Application Installation**](#appinstall)

**[Application performance testing strategies](#apploadtest)**

[**Profiling strategies:**](#profilingstrategies)

[**Results analysis**](#resultanalysis)

[**Learnings**](#learnings)

[**Attached Files Description**](#filedescription)

**[References](#references)**

**Application Details**

Tasteful Server [3] is a web server which provides a multithreaded server architecture for high performance network applications. It is written in C++ and uses the Qt library. You can specify a number of servers with their own handlers. Those servers share a thread pool of user defined size to handle incoming data.

**Application Installation steps:**

The application installation and profiling is done on Ubuntu 14.04.3 LTS. All the steps are with respect to this platform.

Installing prerequisites:

1] Visit Qt downloads page and download a 32-bit or 64-bit Linux installation depending your version of Ubuntu. The installation file can be also downloaded through the command line using wget.

$ wget http://download.qt.io/official\_releases/qt/5.5/5.5.1/qt-opensource-linux-x64-5.5.1.run

2] Install qt:

$ sudo chmod +x qt-opensource-linux-x64-5.5.1.run

$ ./qt-opensource-linux-x64-5.5.1.run

3] Install g++

$ sudo apt-get install build-essential

Note: Just installing the above-mentioned mesa-common-dev kit is not sufficient for more recent Ubuntu versions. Based on this comment in the forum an additional package needs installation. Execute following command:

$ sudo apt-get install libglu1-mesa-dev -y

4] Install CMake:

Adding PPA:

$ sudo apt-get install software-properties-common

$ sudo add-apt-repository ppa:george-edison55/cmake-3.x

$ sudo apt-get update

$ sudo apt-get install cmake

Installing application:

1] Download the source code from GitHub:

$ wget <https://github.com/scheibel/tastefulserver/archive/master.zip>

[I have modified source code to accept user defined number of threads. Modified code is attached in folder tastefulserver-master\_modified]

$ ll master.zip

-rw-rw-r-- 1 avp avp 112969 Dec 10 00:42 master.zip

2] Extract it:

Install extractor if it is not present:

$ sudo apt-get install unzip

$ unzip master.zip

$ ll

drwxrwxr-x 6 test test 4096 Jul 23 03:38 tastefulserver-master

$ pwd

/<path\_to\_source>/tastefulserver-master

3] Build it

$ mkdir build && cd build

$ pwd

/<path>/tastefulserver-master/build

Install Qt 5 development defaults package

$ sudo apt-get install qt5-default

$ sudo cmake ..

[I had received an error at this step:

“Could not find a package configuration file provided by "Qt5Core" (requested version 5.0) with any of the following names"

Solved this using:

$ sudo apt-get install qt5-default]

$ sudo make

$ pwd

/<path>/tastefulserver-master/build

$ ls

CMakeCache.txt cmake\_install.cmake CPackSourceConfig.cmake httpserver install\_manifest.txt Makefile packages source websockets

CMakeFiles CPackConfig-tastefulserver.cmake docs httpsserver libtastefulserver.so output revision

4] Now the package is built. The Tasteful Server library and its header files can be installed in your system using the following command:

$ sudo make install

5] You can now run the application using following commands:

$ cd /usr/local/share/tastefulserver/examples

To start the HTTP Server:

$ httpserver <number of threads>

You can access it in your browser at URL: <http://localhost:8080/>

To start the Secure HTTP Server:

$ httpsserver <number of threads>

You can access it in your browser at URL: <https://localhost:8081/>

=====================

Installing profilers:

\*\*\*\* aprof \*\*\*\*

1] Download it (*aprof 0.2.1*) from https://github.com/ercoppa/aprof/wiki#download

extract it

$ tar -xvf aprof-0.2.1.tar.gz

2] Change directory where profiler has been extracted.

$ cd aprof-0.2.1

$ pwd

/<path>/aprof-0.2.1

3] Run ./autogen.sh to setup the environment (you need the standard autoconf tools to do so, so install automake).

$ sudo apt-get install automake

$ ./autogen.sh

4] Run ./configure, with some options if you wish. The only interesting one is the usual --prefix=/where/you/want/it/installed

$ ./configure

5] $ sudo make

6] Run "make install", possibly as root if the destination permissions require that.

$ sudo make install

7] Try "valgrind ls -l": If this works that means the installation is successful.

\*\* *Important*! Do not move the Valgrind installation into a place different from that specified by --prefix at build time. This will cause things to break in subtle ways, mostly when Valgrind handles fork/exec calls.

\*\*\*\*\*\*\*\* Installing zoom \*\*\*\*\*\*

1] Download Zoom\_3.3.3\_Linux-x86\_64.tar.gz from website:

$ wget http://www.rotateright.com/download/Zoom-3.3.3/Zoom\_3.3.3\_Linux-x86\_64.tar.gz

2] Install OProfile and Linux Generic Tools

$ sudo apt-get install oprofile linux-tools-generic linux-headers-`uname -r`

3] Extract and run install.sh and follow on screen steps (next -> next -> ....)

**Application performance analysis techniques used**

Application Under Test (AUT) is a web server, so for testing its performance, I decided to measure its response rate to the maximum number of connection requests it can serve at given time. I have used below techniques to load test the server.

1] httperf:

*httperf* is a tool for measuring web server performance. It provides a flexible facility for generating different HTTP workloads and for measuring server performance.

httperf has different parameters which can be configured. I have used following parameters:.

*--num-conns=N*:

it specifies total number of connections to create.

**--***rate****=****X:*

Specifies the fixed rate at which connections or sessions are created

***--****port****=****N:*

This option specifies the port number *N* on which the web server is listening for HTTP requests.

***--****timeout****=****X:*

Specifies the amount of time *X* that httperf is willing to wait for a server reaction. The timeout is specified in seconds. This timeout value is used when establishing a TCP connection, when sending a request, when waiting for a reply, and when receiving a reply. If during any of those activities a request fails to make forward progress within the alloted time, httperf considers the request to have died, closes the associated connection. By default, the timeout value is infinity.

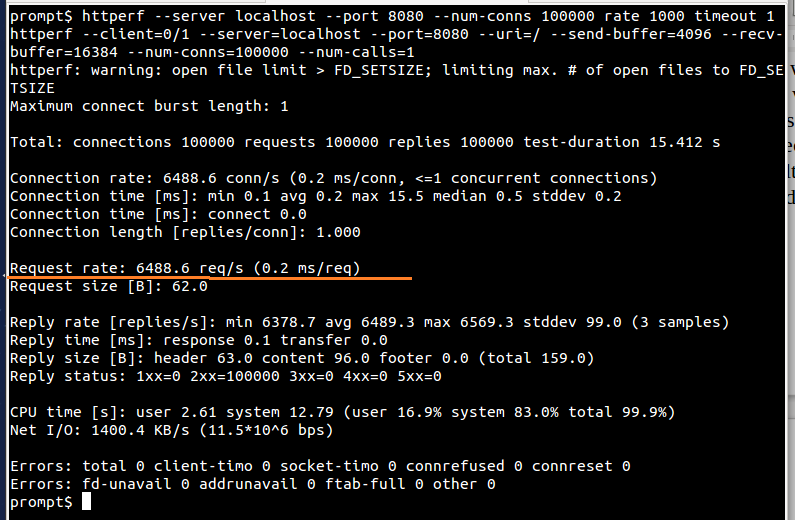
After observing several connection request results, I found out that average time required for each connection was ~ 0.3 ms. To be on safer side, I have used timeout=1 in all my experiments.

Here is the sample output of the command:

$ httperf --server localhost --port 8080 --num-conns 10000 rate 1000 timeout 1

As our server is installed on the same machine, I provide localhost as the sever address and 8080 is the port on which the server is listening to the requests.

Output:



**Httperf command output**

2] Apache Benchmark (ab):

*ab* is a tool for benchmarking HTTP web servers. It gives an impression of how our server installation performs for different workloads. This especially shows how many requests per second your web server is capable of serving. ab command takes different input parameters as explained below:

*-c* concurrency

Number of multiple requests to perform at a time. Default is one request at a time.

***-****k*

HTTP KeepAlive means to perform multiple requests within one HTTP session. This option enables the HTTP KeepAlive feature, Default is no KeepAlive.

*-n* requests

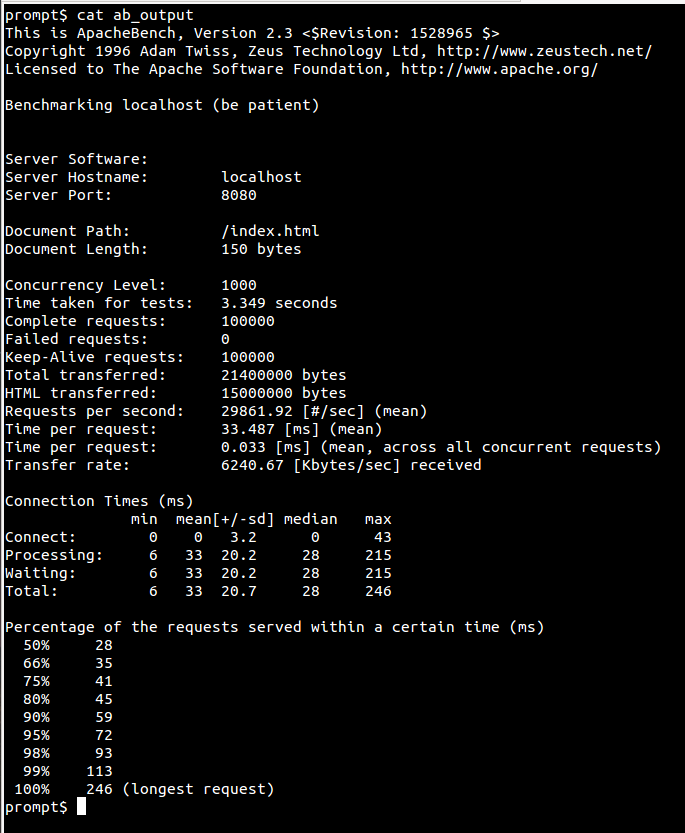
Number of requests to perform for the benchmarking session. The default is to just perform a single request which usually leads to non-representative benchmarking results.

Here is the output of the sample command:

$ ab -k -n 100000 -c 1000 <http://localhost:8080/index.html> > output

Here <http://localhost:8080/index.html> is the homepage of the web server installed.

Output:



**ab command output**

3] Parallel downloads using threads:

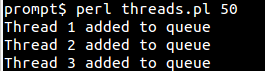
I have written a Perl script, *threads.pl* which parallely downloads index.html page from the web server localhost:8080/index.html. This script takes number of threads as an input parameter and each thread downloads the page using *wget* utility multiple (50) times. As these threads run in parallel, several connection requests are sent to the server. Thus, this script helps us to see how server responds to multiple requests.

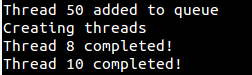
Command to execute the scripts:

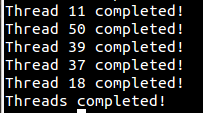
$ perl threads.pl <number of threads>

Example: perl threads.pl 50

Here is the sample output:







**Profiling Strategies:**

Profiling [5]

In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

I have used following profilers:

1] Aprof[4]

Aprof is a Valgrind tool for performance profiling which helps developers discover hidden asymptotic inefficiencies in the code. From one or more runs of a program, aprof measures how the performance of individual routines scales as a function of the input size. aprof creates a .aprof file for every thread application created and stores the profiling results in it. This file can later be read by a tool aprof-plot[] which visualises the performance profiles generated by aprof.

I have used aprof to find the cost (number of executions) of each method and thus obtain top 10 methods are taking maximum time.

You can run this tool using valgrind tool as:

$ valgrind --tool=aprof httpserver 5

where,

--tool specifies which Valgrind tool you are using

httpserver is application binary and

5 is number of threads application should create.

2] Callgrind[8]

callgrind is a Valgrind tool, which gives caller and callee graphs for methods executed. I used this tool to find the flow of method calls once the application starts. Result of callgrind is stored in callgrind.out.<pid> file. This file can be read with the help of KCachegrind[]. KCachegrind presents visual summary of all the callgrind output.

Callgrind can be run as:

$ valgrind --tool=callgrind httpserver 4

Kcachegrind can be used to visualise the summary as:

$ kcachegrind callgrind.out.6348

where callgrind.out.6348 is the output of previous callgrind command.

Note: You can install Kcachegrind using the command:

$ sudo apt-get install kcachegrind

3] DRD[9]

drd is another Valgrind tool which is used for detecting errors in multithreaded C and C++ programs. This tool works for applications which implements threads using POSIX threads libraries. This tool also gives you methods which are holding locks for longer time. I used this tool to find methods which may cause lock contention. Output of drd specifies time for which a method holds the lock. If you specify threshold time, drd lists methods which hold the lock for longer duration than the threshold.

Initially I listed all the methods with threshold value = 1ms for different input combinations (100 httperf connection requests and downloads using 50 parallel threads of threads.pl) and I observed that minimum time every method was holing lock for was ~ 10ms, so for experimental purpose, I have kept threshold = 10.

This tool can be run as:

$ valgrind --tool=drd -v --exclusive-threshold=10 --error-limit=no httpserver 4

where,

-v gives count of detected and suppressed errors

--exclusive-threshold prints an error message if any mutex or writer lock has been held longer than the time specified in milliseconds. This option enables the detection of lock contention. --error-limit=no specifies that all the errors will be reported

This tool prints entire output and errors reported on the STDOUT. To capture this in a file, I have used *script* command which captures the output in a file.

$ script -c 'valgrind --tool=drd -v --exclusive-threshold=10 --error-limit=no httpserver 4' -f drd\_httperf\_conn\_100000\_rate\_1000\_4\_httpserver

where,

-c specifies command to be executed and

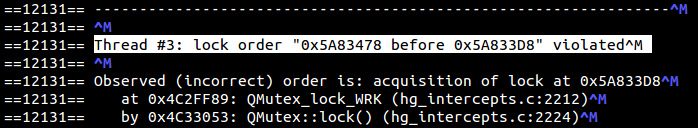
-f specifies output file name

4] Helgrind[10]

Helgrind is a Valgrind tool for detecting synchronization errors in C and C++ programs that use the POSIX pthreads library. Helgrind can detect potential deadlocks arising from lock ordering problems and data races occurring without proper locking or synchronization. For few inputs, I observed that locking order followed by threads created by the application was inconsistent. Using helgrind's output log, I found out that few threads violated locking order and thus made some conflicting writes to shared variables.

Helgrind can be run as:

$ script -c 'valgrind --tool=helgrind --error-limit=no httpserver 4' -f helgrind\_100\_perl\_threads\_4\_httpthreads

This is the snippet of log file *helgrind\_100\_perl\_threads\_4\_httpthreads.* This file is attached and is present in the results directory. From the log, we can see that Thread 3 has acquired lock on some resource at the position 0x5A833D8 while that resource is being released at location 0x5A83478

**Results Analysis**

**1] Control Flow:**

Below diagram illustrates the procedure flow for the application when httpserver has started.

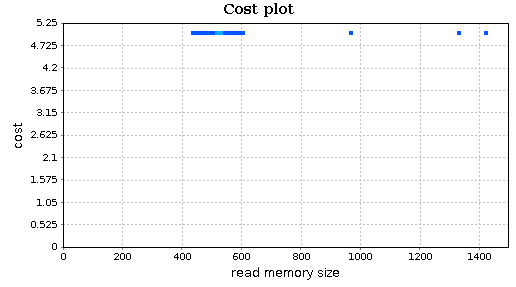
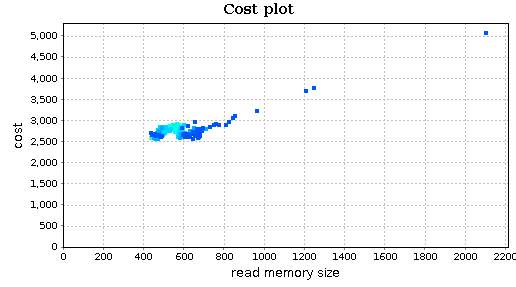
****

**Procedure Invocation Flow Diagram**

**2] Aprof Results:**

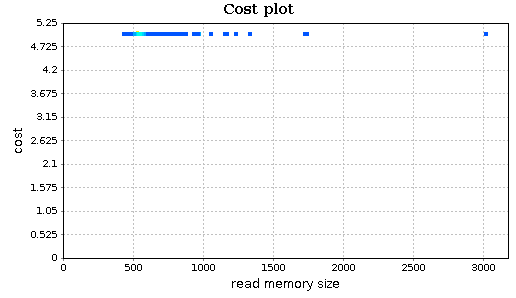
Below are the Read Memory Size Cost Plot for CreateSocket() method of HttpSocket class. For every connection request, a new socket needs be created and under heavy load, this method was the most frequently called method. So, I decided to study RMS cost of this method for varied workload and plotted the results. This method is called by startTask() which handles queued operations.

*RMS Plot for 100 Connection threads:*

****

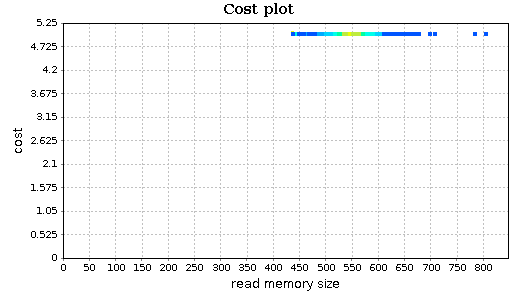
**4 application (httpserver) threads 8 application (httpserver) threads**

*RMS Plot for 200 Connection threads:*

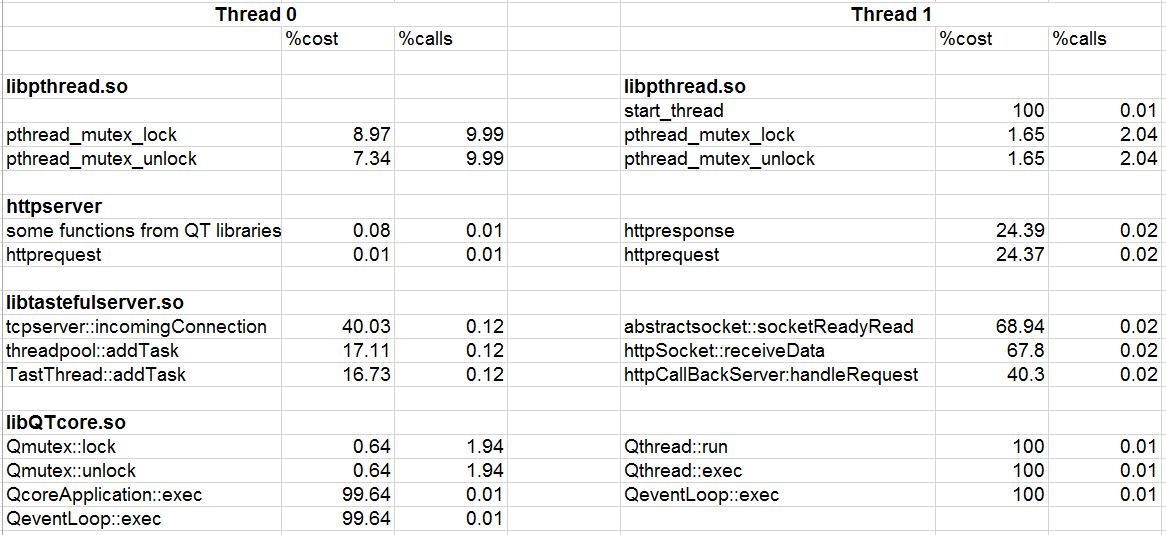
****

**4 application (httpserver) threads 8 application (httpserver) threads**

*RMS Plot for 500 Connection threads:*

** 4 application (httpserver) threads 8 application (httpserver) threads**

**Read Memory Size Cost Plot For HttpSocket::CreateSocket() Method For Various Workloads**

****

**Method Calls and Costs Obtained After 1000 Connection Threads And 4 Httpserver Threads**

**3] Thread Handling Methods**

Task.cpp and ThreadPool.cpp contain methods related to synchronization and multithreading.

**4] Bottlenecks and Inefficiencies:**

After trying several profilers and analyzing them finally I found a good profiler which can help me profile the application the way I was looking for. DRD, as we have seen earlier is a Valgrind tool and helps find hotspots in the application. DRD instruments the application and tries to find the methods which are holding locks for longer time than usual and thus responsible for lock contention which in turn creates bottlenecks.

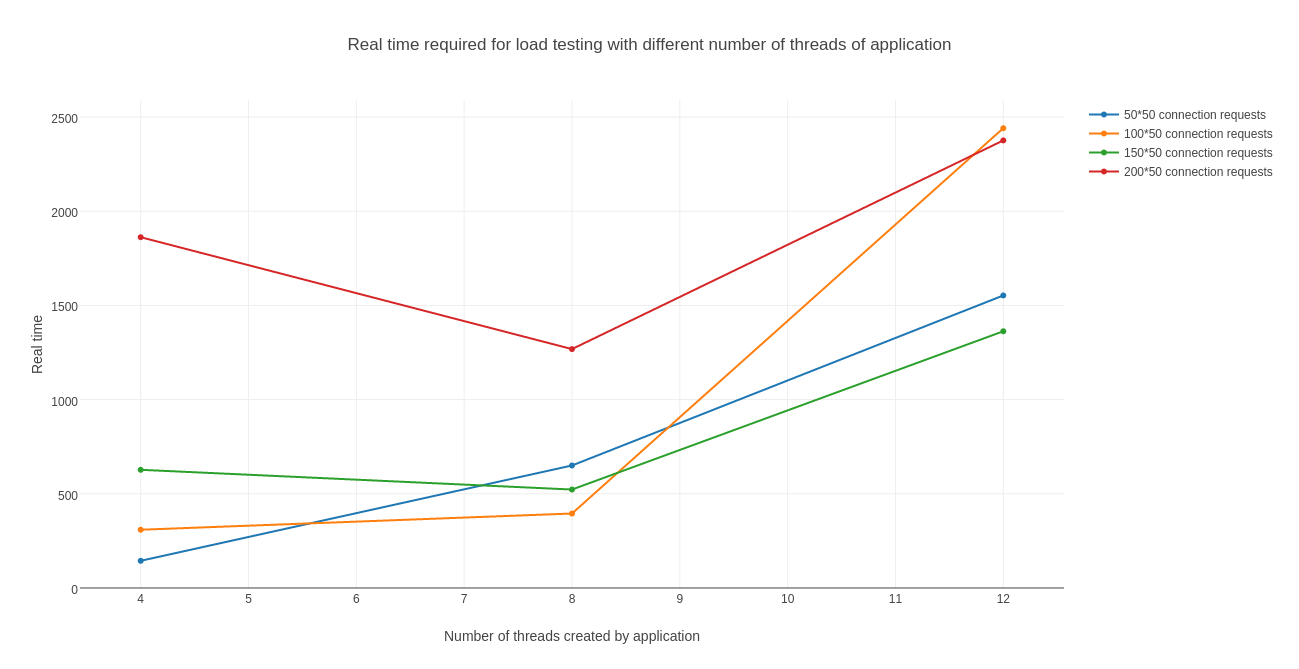
I have written a script *drd\_automate.sh* which first starts profiling our binary (httpserver in this case) and runs it in the background. Pid of this process is stored so that we can kill it when its purpose is done. Next, *threads.pl* script is executed which tries to download server homepage multiple times in parallel. Thus we can observer server’s performance when dealing with multiple simultaneous connections. Once threads.pl script is completed, we kill the Valgrind process using its stored pid. For the next iterations, I change number of threads created by application as well as number of threads which tries to access the server.

For each iteration, output of every command is printed to file using *scipt* utility. Format of output file is specified as below.

DRD log file: Drd\_<no. of perl threads>\_with\_threshold\_10\_<no. of httpserver threads>

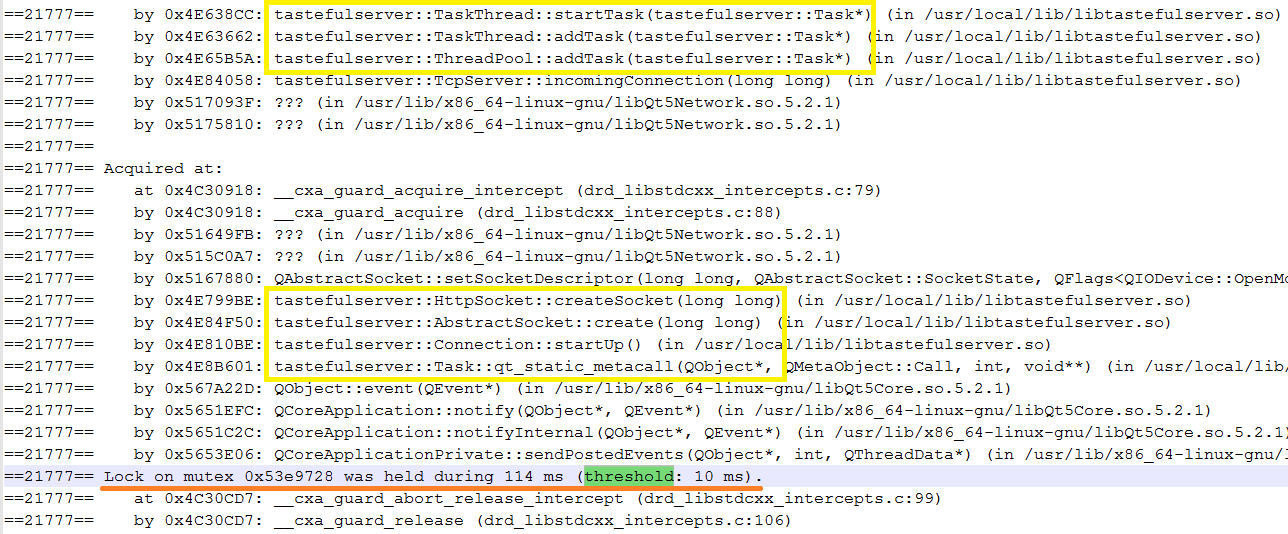
Time log file: time\_<no. of perl threads>\_<no. of httpserver threads>

After the script *threads.pl* completed successfully. I plotted a graph of number of threads vs real time taken by the Valgrind process to complete. I observed that as the number of threads increased, time taken by the process to finish increased instead of decreasing. I was surprised by the results, so I went through the logs created and found some interesting points.

**Plot: Number of threads created by application vs Real time taken by process to finish in seconds**

When I was going through the drd logs, I found out that few methods were holding locks for longer time than they usual. When I say usual, I compared this with drd logs of same workload with lesser number of threads.

For illustration, I have added snippets of both the logs.



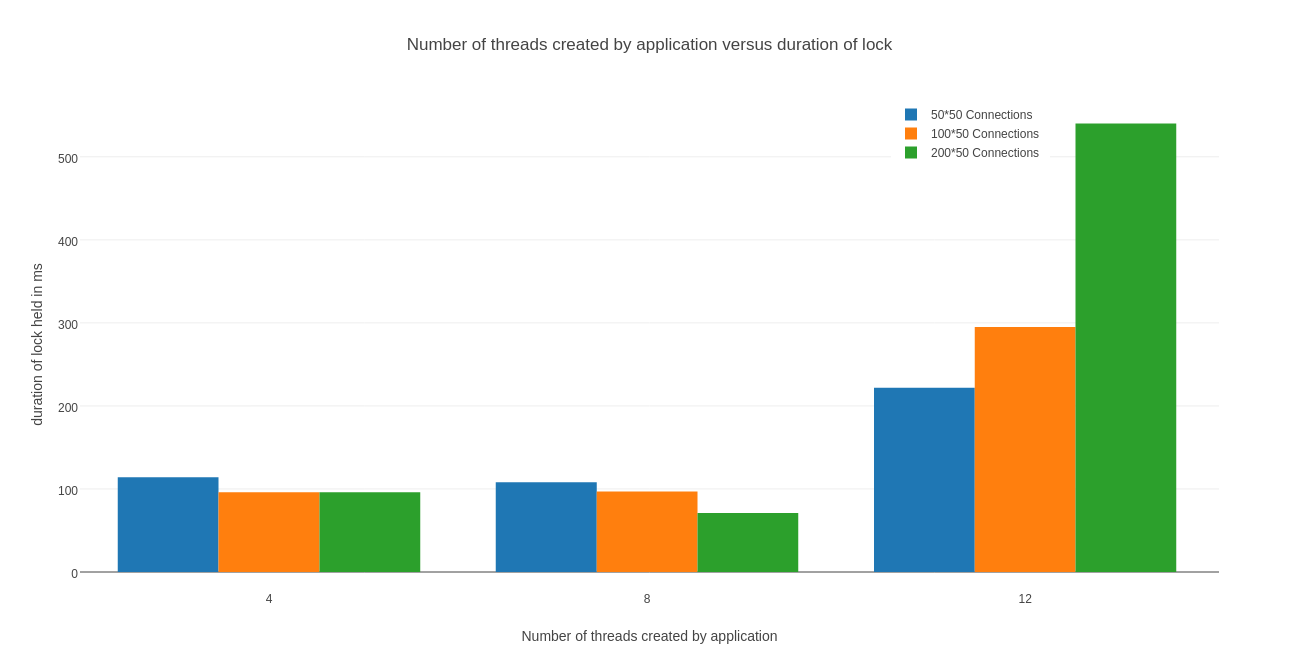
**Snippet of drd log file with 50 connection threads (from threads.pl) and 4 httpserver (application) threads**

From above file, we can see that startTask() and addTask() hold the lock for 114 ms, this log contains.



**Snippet of drd log file with 50 connection threads (from threads.pl) and 12 httpserver (application) threads**

From above snippet, we can observer that startTask() and addTask() hold the lock for 222 ms, this log contains.

****

**Plot for duration of lock held in milliseconds vs number of threads created by application for three different workloads**

From above graphs, we can observe that, duration of lock held by methods was linear when number of locks were limited to 8. But when number of threads were increased to 12, lock duration was significantly increased. This shows that when the number of threads created by application increase beyond 8, application is not able to handle the synchronization and locking ordering properly.

**Problems Faced**

**1] glibc error:**

While performing load testing of the server using apache benchmark (ab) tool, I encountered an error. I had issued following command:

$ ab -k -n 1000000 -c 1500 -s 3000 <http://localhost:8080/index.html>

Error message: *(process:8302): GLib-WARNING \*\*: poll(2) failed due to: Invalid argument*

After this error, server became unresponsive and ab was not able to connect send anymore requests to the server. After reading about this error I found out that the problem was due to limit on maximum number of sockets desciptors a process can held at a given time. I checked the limits for my system using *ulimit -n* command.



-c option of ab specifies number of multiple requests to make at the same time, I was trying to create more sockets (1500) than allowed by system.

Resolution:

- Default limits given by system on maximum number of descriptors can be overridden by modifying /etc/security/limits.conf file in Ubuntu.

- I added below lines to the /etc/security/limits.conf file.

avp soft nofile 10000

avp hard nofile 100000

- Save the file and reboot.

I ran the same command and it finished without any error. I tried with -c 2000 and -c 3000, they were working fine.

**2] Constant number of threads**

Initially, I found some strange behavior of the application. For all the load testing methods described earlier (ab, httperf, threads.pl); irrespective of the workload (I.e. number of connection requests, connection request rate, number of threads parallely accessing server homepage), number of threads created by application were constant (=2) in all the cases.

To find out number of threads created by process, I used two approaches:

- Check ps output:

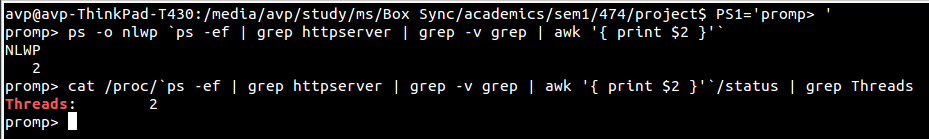
$ ps -o nlwp `ps -ef | grep httpserver | grep -v grep | awk '{ print $2 }'`

Where, ps -o nlwp <pid> gives number of threads created by process given by pid.

`ps -ef | grep httpserver | grep -v grep | awk '{ print $2 }'` gives pid of process httpserver

- Status file information: In Ubuntu, /proc/<pid>/status stores all information about a running process including number of threads created.

$ cat /proc/`ps -ef | grep httpserver | grep -v grep | awk '{ print $2 }'`/status | grep Threads

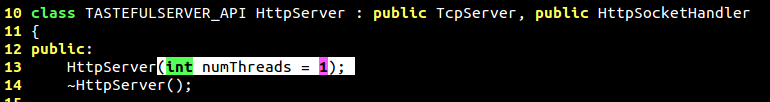


I went through the code where threads are created, I found out that the AUT uses TaskThread and ThreadPool libraries to manage threads. AUT uses idealThreadCount() method provided by Qthread library to determine maximum number of threads to be created.

*int Qthread::idealThreadCount():*

Returns the ideal number of threads that can be run on the system. This is done querying the number of processor cores, both real and logical, in the system. This function returns -1 if the number of processor cores could not be detected [11].

For my system configuration, this method returns 4 but number of threads was being set to 1 by default by the calling method. So total number of threads, including main thread for the application were 2.



Snippet of file *source/tastefulserver/include/tastefulserver/HttpServer.h*

Initially, for long time I was stuck with constant number of threads problem. I used callgrind tool which provided caller and callee graph which helped me understand the procedure call sequence. I figured out the routine which was setting number of threads to 1 by default. Then I modified the source to take number of threads as command line parameter. Now I was able to start the server with user defined number of threads and the next process began.

**Learnings**

1] Right tool makes the difference

Initially I was randomly selecting profilers and trying to find some (any type of) bottleneck in the application. But after discussing progress with Bikas, I understood that major focus should be on thread locks contention. After that discussion I started looking for multithreading related profilers. Earlier, I spent a lot of time playing with different profilers, especially aprof and see how do they work, what are their features.

I came across DRD vary late. Had I found this tool earlier, I could have done more in-depth analysis and possibly could have tried to fix the problems.

2] Continuous feedback on work is important

From the start of the project, I was running the program as it is and was trying to figure out why the number of threads is always constant. Was trying various load testing techniques to see what type of load will increase the number of threads. I was under the impression that we had to profile the original code (the code author of the application has written). But when I discussed this with problem with Bikas and Professor Mark, they suggested me to change the application source code and

3] Importance of Documentation at Given Time

You should document your work immediately after you complete it, in worst case. If you have better idea of what the outcome will be, better write it (or at least take notes of them) before you do it, so that at last moment you need not worry about how you did that and if you are missing anything. In summary, basic design should be prepared before actually starting the work.

**Attached Files Description**

1. **Scripts Directory**
   1. drd\_automate.sh
   2. threads.pl
2. **tastefulserver-master\_modified Directory**

Modified source code to accept number of threads as command line parameter

1. **tastefulserver-master\_original.zip**
   1. Original source code downloaded from GitHub
2. **Results\_logs directory**
   1. Contains results and logs which are used in the report (for reference purpose)

**Note:**

1] Only application source code (original and modified) are attached. Dependencies as well as source to obtain are clearly stated in installation steps. Actual installation files are not attached because of larger file size.

2] All the experiments performed are on my Ubuntu 14.04.3 LTS, 64-bit, Kernel 3.19. All the steps mentioned work perfectly on my system mentioned above. If any discrepancy is found, I request you to please contact me.

3] drd\_automate.sh produces 12 drd log files and 12 time files. DRD log files are very large (~7-9 MB each). So I have attached only four for illustration purpose.

**References**

1. UIC Logo: http://logos.uic.edu/DOWNLOAD.CGI?document=COL.ENG.CSCI.LOCKB.SM.RED.PNG
2. Profiling image: http://www.layer7.co.za/app\_profiling.html
3. Application: https://github.com/scheibel/tastefulserver
4. Aprof: https://github.com/ercoppa/aprof
5. Profiling: https://en.wikipedia.org/wiki/Profiling\_(computer\_programming)
6. Apache Benchmark: https://httpd.apache.org/docs/2.2/programs/ab.html
7. httperf: http://linux.die.net/man/1/httperf
8. Callgrind: http://valgrind.org/docs/manual/cl-manual.html
9. DRD: http://valgrind.org/docs/manual/cl-manual.html
10. Helgrind: <http://valgrind.org/docs/manual/hg-manual.html>
11. QThread Documentation: http://doc.qt.io/qt-4.8/qthread.html#idealThreadCount