Input-Sensitive Profiling LinuxDC++

Pavan Kumar Kothagorla, Praval Jain, Sandeep Kumar Anil Kumar, Shiwangi Singh

Introduction:

Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. It is done by instrumenting either the application under test (AUT) source code or its binary executable form by using a tool called profiler. **Instrumentation** is a technique that inserts extra code into the AUT to collect runtime information. **Profiler** is a tool that inserts instructions or codes into the AUT to obtain frequency, memory usage and elapsed execution time of method calls.

Profiling is broadly of two types: standard profiling and Input-sensitive profiling. In **standard profiling**
methodology the input to an application is given as a concrete set of values. Profilers that are based on standard methodology are ubiquitous and easy to use; however, their key weakness is based on the assumption that all input data is available in advance, its size is small and finding bottlenecks is orthogonal to the type and the size of the input data. This assumption reduces the effectiveness of profiling for solving performance problems.

Input-sensitive profiling departs from the standard profiling methodology by inferring the size or the type of the input that can pinpoint performance problems in a software application.[7]

The application under test for this project is DC++. It is an open source, peer-to-peer file-sharing client for Windows for the Direct Connect /Advanced Direct Connect network. Direct Connect allows people to share files over the Internet without restrictions or limits. DC++ comes with an integrated Firewall and router support and has functionalities such as multi-hub connections, auto-connections and resuming of downloads. DC++ is free software, licensed under the GNU GPL 2 and has been programmed in C++. [1][2]

Choice of Profiler:

The profiler implemented for profiling DC++ in this project is **aprof**. It is a Valgrind tool for performance profiling designed to help 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, yielding clues to its growth rate and to the "big-O" of the program. The info provided by aprof is a lot more precise: for example you can easily draw a chart and evaluate the trend growth rate (a linear trend as expected). A traditional profiler can only give you aggregate info because it does not know the input size of a routine invocation and therefore it treats all the routine calls as executed on the same unknown input size.[5][6]

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile programs under test in detail. Valgrind is a programming tool for memory debugging, memory leak detection, and profiling. It is in essence a virtual machine using Just-In-Time (JIT) compilation techniques, including dynamic recompilation. Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a

temporary, simpler form called Intermediate Representation (IR), which is a processor-neutral, SSA-based form. After the conversion, a tool is free to do whatever transformations it would like on the IR, before Valgrind translates the IR back into machine code and lets the host processor run it. It recompiles binary code to run on host and target CPUs of the same architecture. It also includes a GDB stub to allow debugging of the target program as it runs in Valgrind, with "monitor commands" that allow one to query the Valgrind tool for various sorts of information.[3][4]

Description of the Application Under Test (AUT):

The application under test for this project is **DC++**. It is an open source, peer-to-peer file-sharing client for Windows for the Direct Connect /Advanced Direct Connect network. Direct Connect allows people to share files over the Internet. DC++ comes with an integrated Firewall and router support and has functionalities such as multi-hub connections, auto-connections and resuming of downloads. DC++ is free software, licensed under the GNU GPL 2 and has been programmed in C++. [1][2]

Source code files:
Total lines of code: 81,248
Lines of code: 62,784

Language Breakdown

Language	Code Lines	Comment Lines	Comment Ratio	Blank Lines	Total Lines	Total Percentage
C++	53,668	7,617	12.4%	10,608	71,893	88.5%
XML	8,744	17	0.2%	5	8,766	10.8%
Python	237	35	12.9%	87	359	0.4%
C	95	36	27.5%	26	157	0.2%
Make	40	21	34.4%	12	73	0.1%
Totals	62,784	7,726		10,738	81,248	

Figure 1. Language Breakdown of linuxDC++ [10]

Our Approach for profiling the AUT:

Input Sensitive Profiling in 4 phases

PHASE 1

Description of our approach

We have collected the .aprof files for various inputs for the preliminary stage. These inputs were connecting to users from a hub, downloading files from the user and connecting to multiple users. In the first phase, we analyze the top N methods sorted by their cost in the descending order. The following table shows the list of top 6 methods gathered from several .aprof files generated.

List of analyzed methods:

Method names: -	Cost
Hub::updateStata_gui	26,869,499
BufferedSocket::threadRead	15,803,993
TreeView::sizeDataFunc	5,895,415
MainWindow::on	1,114,222
UserConnection::on	58,867

Table 2: Cost and Top 5 methods for Connecting to Multiple Users

Choice of Plots and Understanding them:

The files generated by the aprof profiler can be analyzed by using aplot, which shows plots of how the method's cost of execution varies with input size. To analyze the methods we got, we choose two of the plots generated by aplot.

1) Cost Plot:

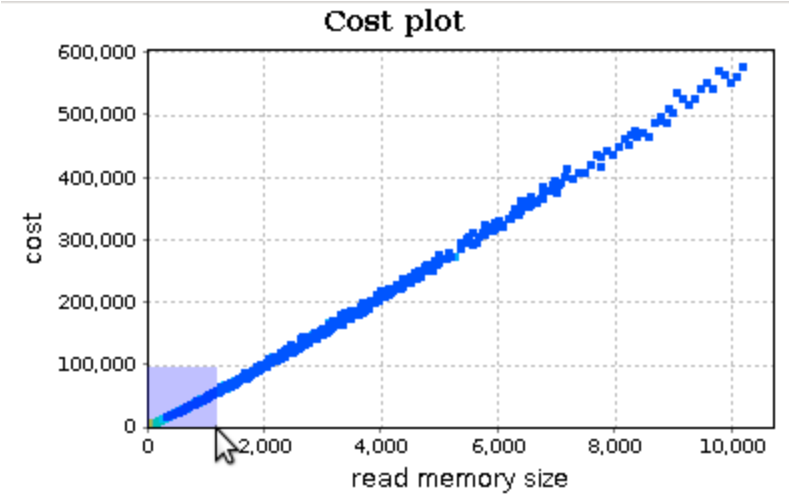
In cost plot, X- axis represents the size of input on which the method is invoked. So it can be seen as an estimate of input size. On Y-axis, execution cost metric. In case of aplot, the execution cost metric is count of no of basic blocks executed. The valgrind profiler translated the code of AUT in to small fragments, which are called basic blocks and then instruments. Hence Y-axis can be seen as an estimate for execution time.

Example:

To understand it better, let us profile a simple code, quickSort

```
int v[10000];
srand(time(0));
int n = sizeof(v)/sizeof(int);
int j = n, i = 0;
for (j = 0; j < n; j += 10000) {
        for (i = 0; i < j; i++)
                v[i] = rand() % (j+1);
        quickSort(v, 0, j-1);
}
```

We are invoking quickSort multiple times with increasing input. The cost plot for this is:

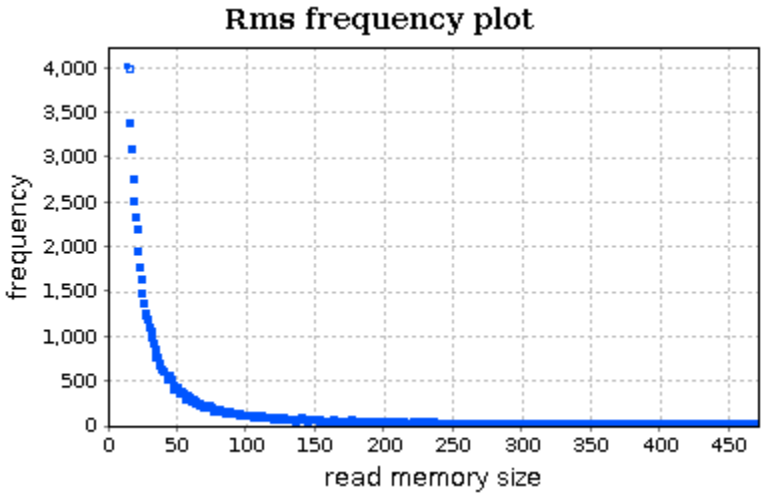


It can be observed that the execution time(Y-axis) does not increase in polynomial time with respect to input size(X-axis)**.** Infact, it seems to vary linear or in n*log(n).

2) RMS frequency plot:

This plot represents the size of input on which the method is invoked and frequency of access of the basic block for that input.

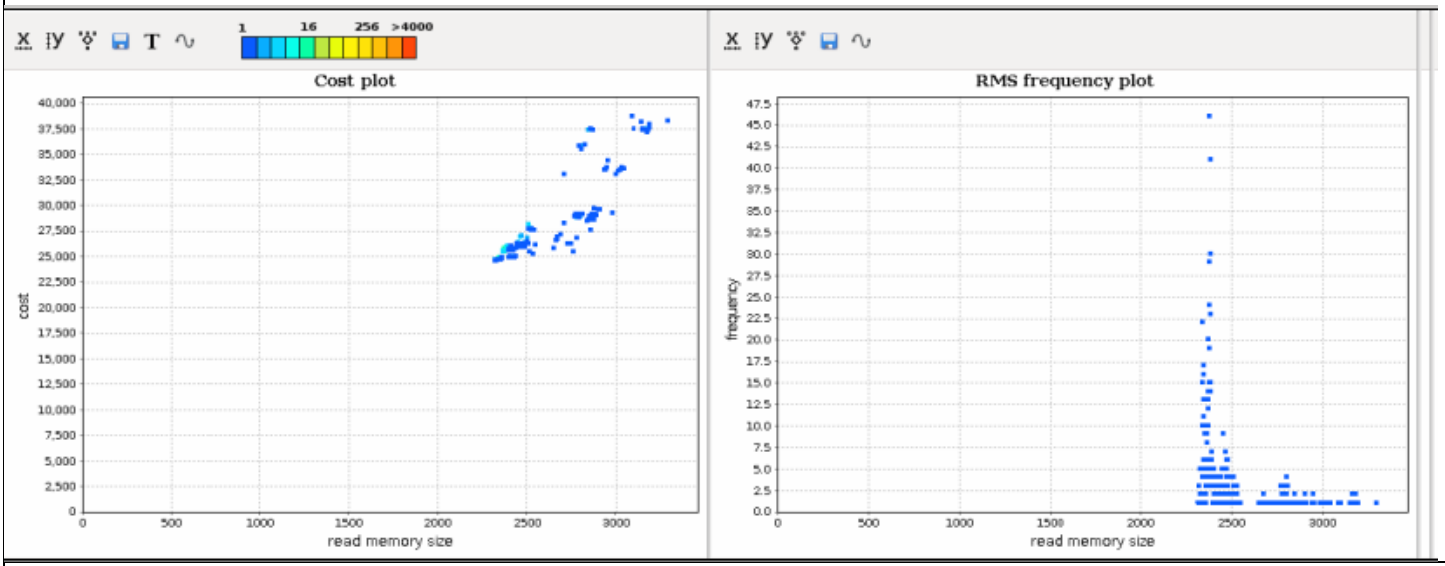
For the quickSort method that is analyzed in previous section, the RMS Frequency plot is



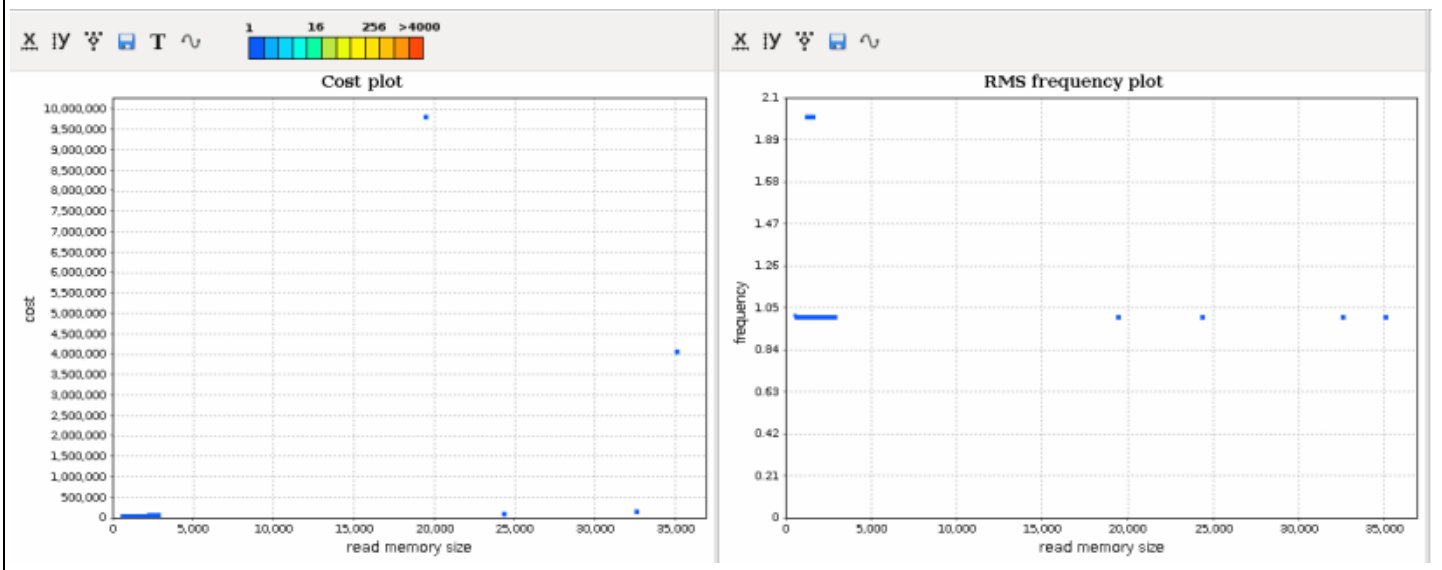
this confirms our observation made in previous section that the execution varies linear or in n*log(n) because , the method is invoked many times on relatively small input sizes and as input size decreases the frequency drops to almost zero. This is expected because quickSort chunks the input in to smaller inputs using divide-and-conquer approach and hence is invoked many times on small input sizes.

Cost plots:

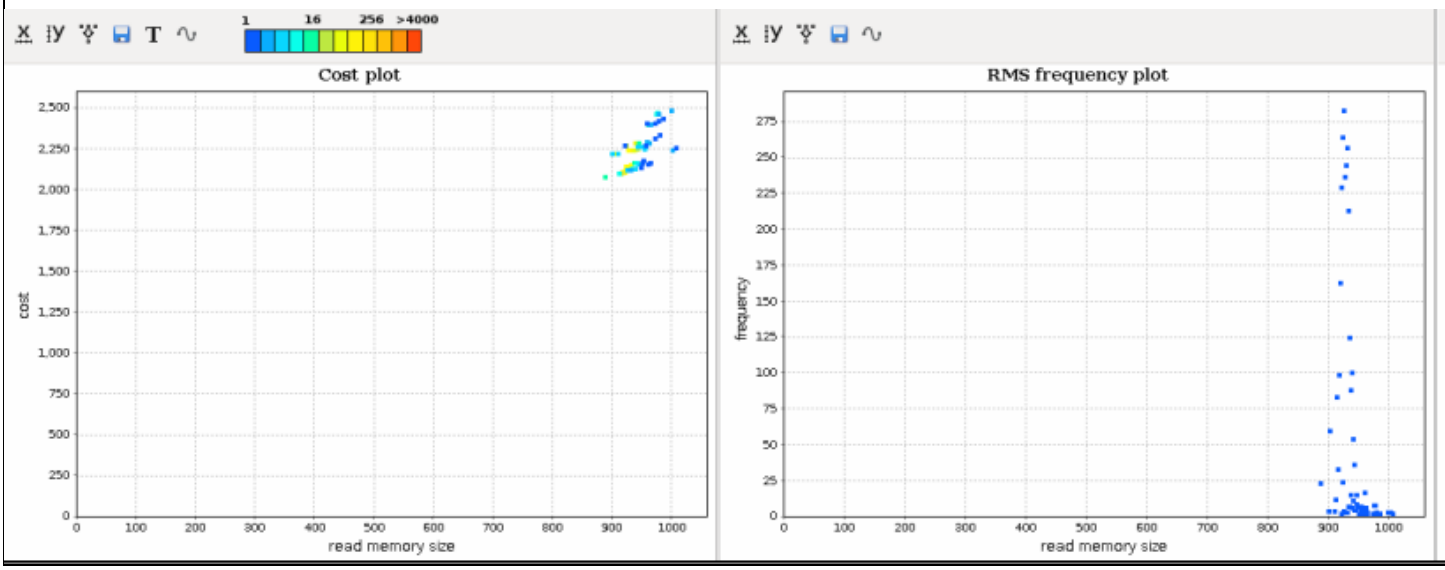
Hub::updateStata_gui



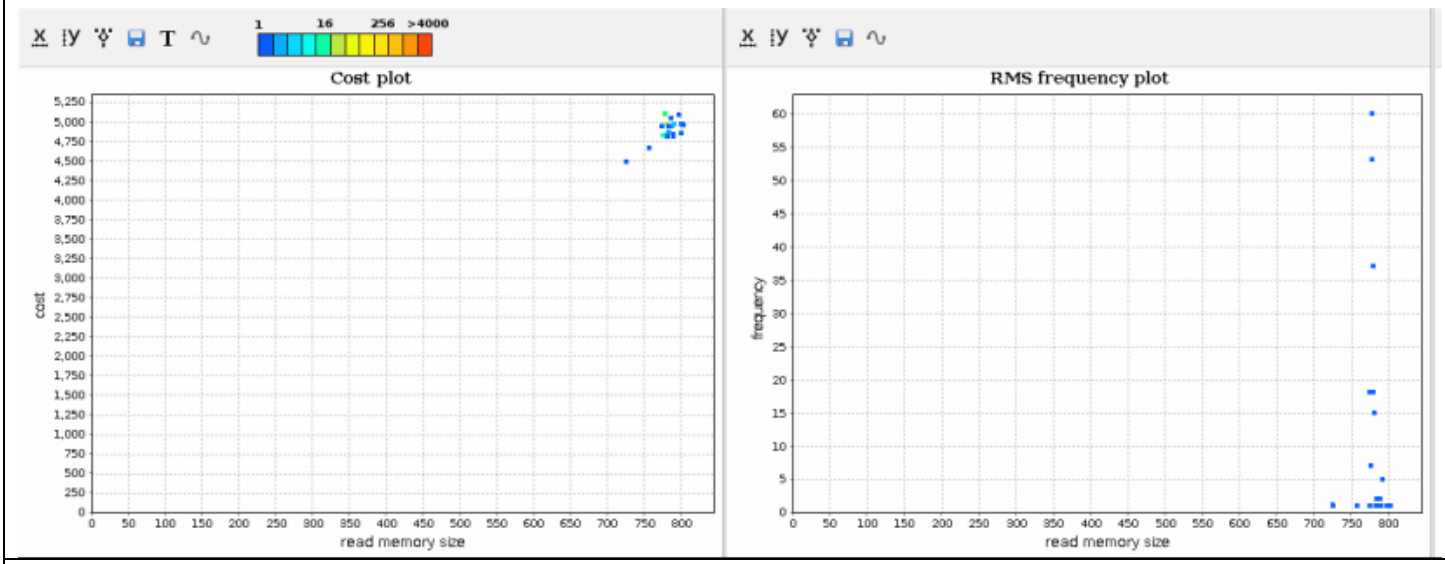
BufferedSocket::threadRead



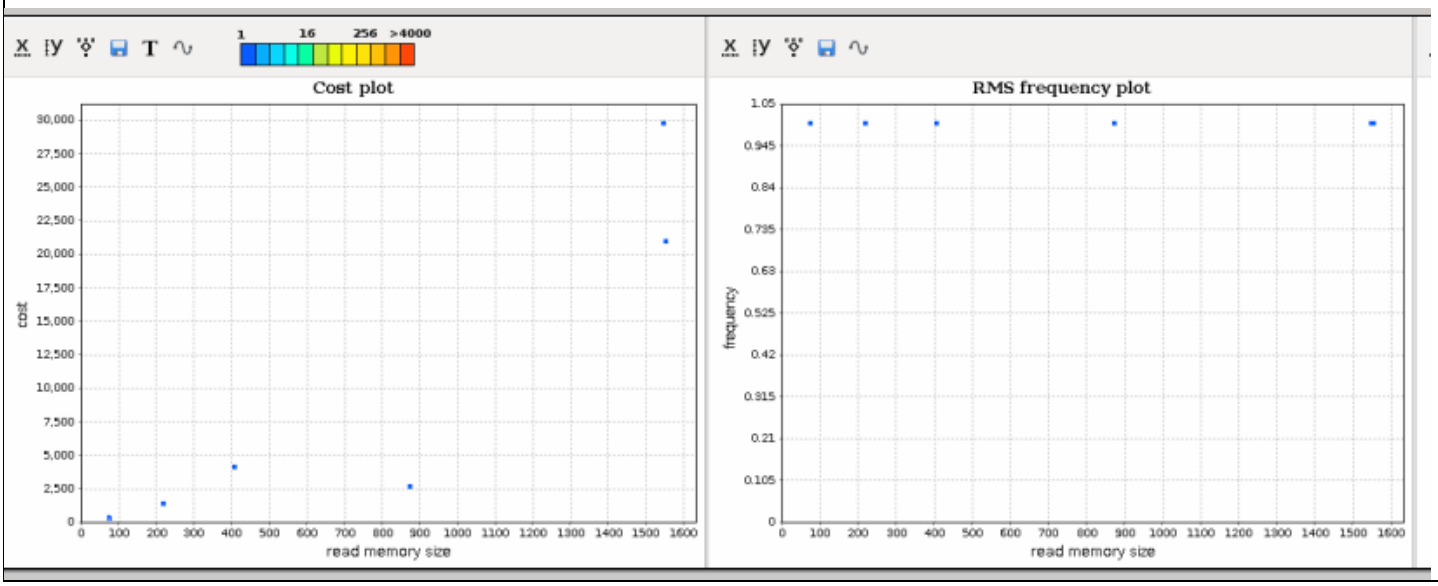
TreeView::sizeDataFunc



MainWindow::on



UserConnection::on



Observations:

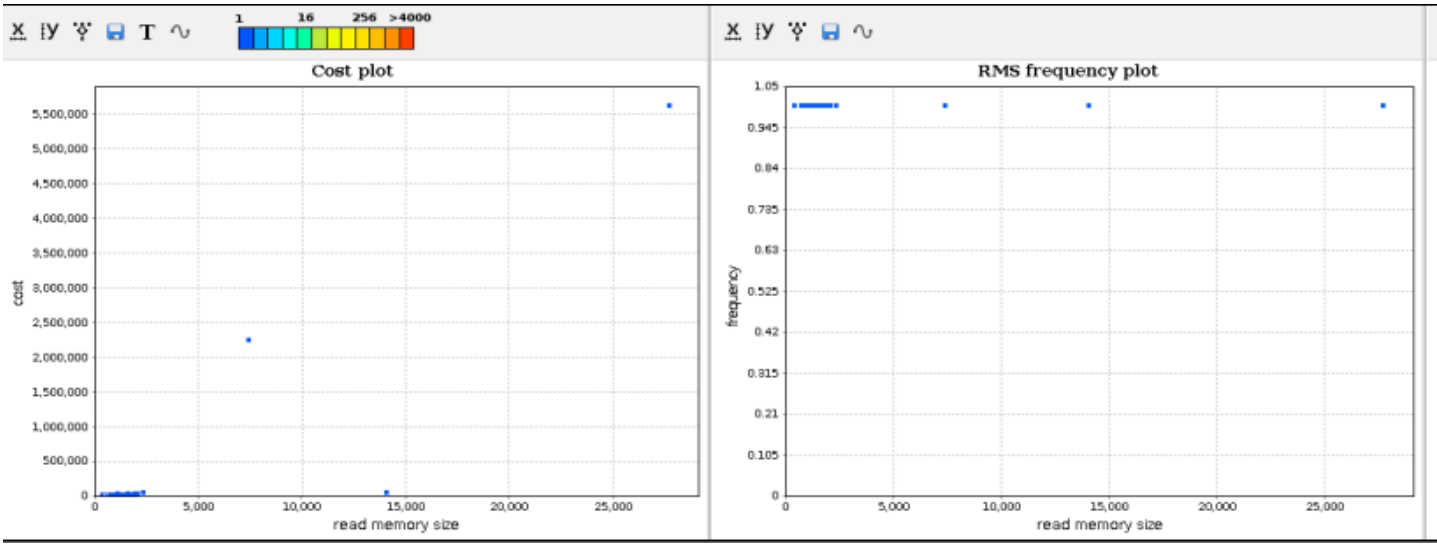
When we plot the graphs for methods which are sorted in decreasing order order of their cost, we observed a pattern that as we move down, the methods were executed on smaller input size and they are taking lesser execution time than previous method (which has higher cost).

PHASE 2

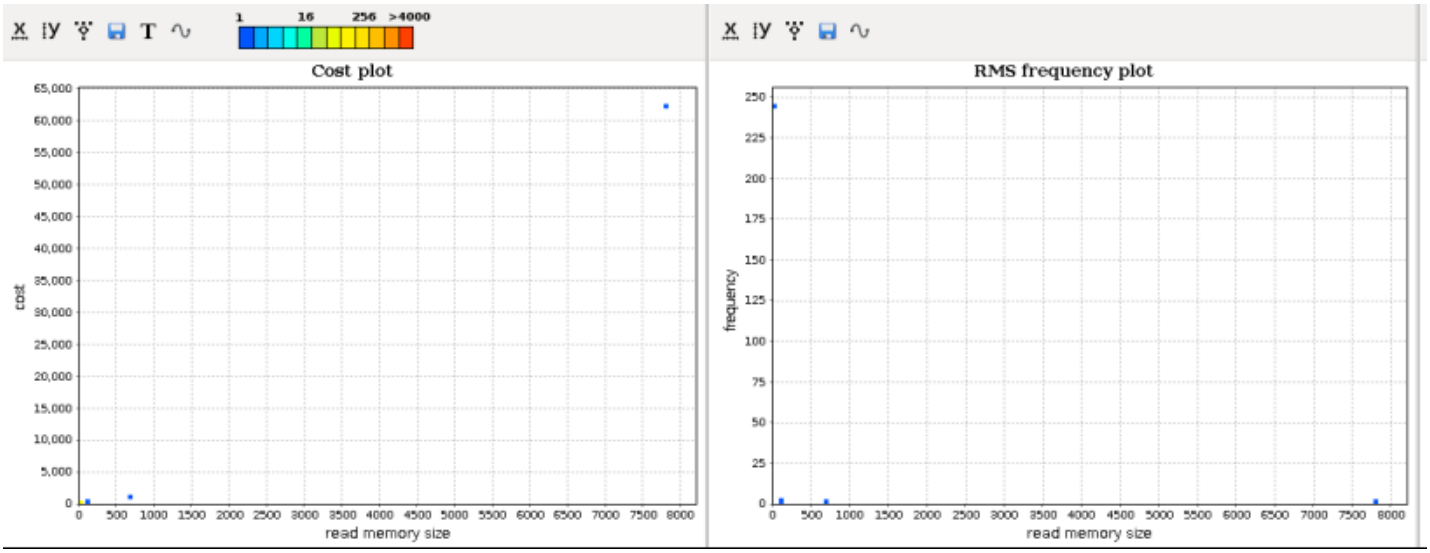
In Phase 2, we instrumented our application with various inputs like downloading single file, downloading multiple files, connecting to multiple users etc… We observed an interesting pattern of how each method and their statistics were listed in the files, which are generated by profiler. Hence we automated the laborious task of finding methods which are called from multiple threads. The code to automate this task was implemented by reading method name from .aprof file and recursively searching for the definition of that method in the code base of our application. Once the definition is found, then the keywords like thread, lock, unlock etc.,

were searched. If a match is found, then we manually crosschecked those method's definitions to make sure that they actually are being called from multiple threads. There were approximately 370 methods in total which were called for downloading files. When this list of methods is given as input to our .java code, it gave a list of 4 methods which are called from multiple threads. We crosschecked these methods and they are actually called from multiple threads.
The list of these methods are
(Please note this is not an exhaustive list).

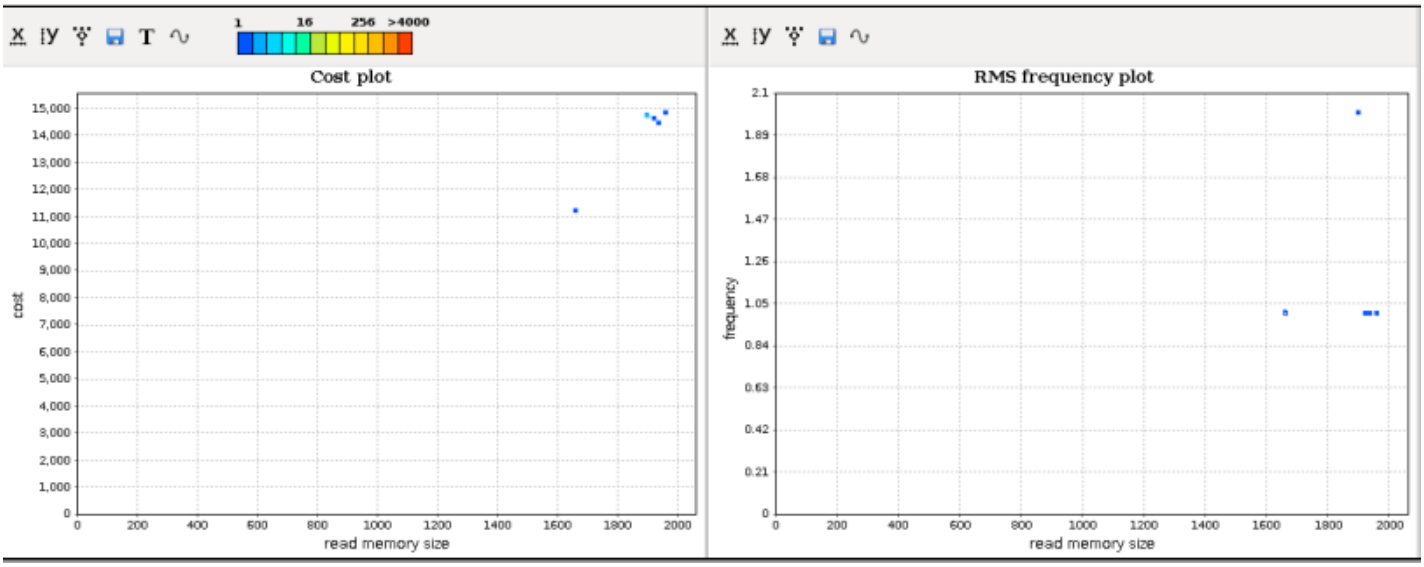
Method name: -
BufferedSocket__threadRead



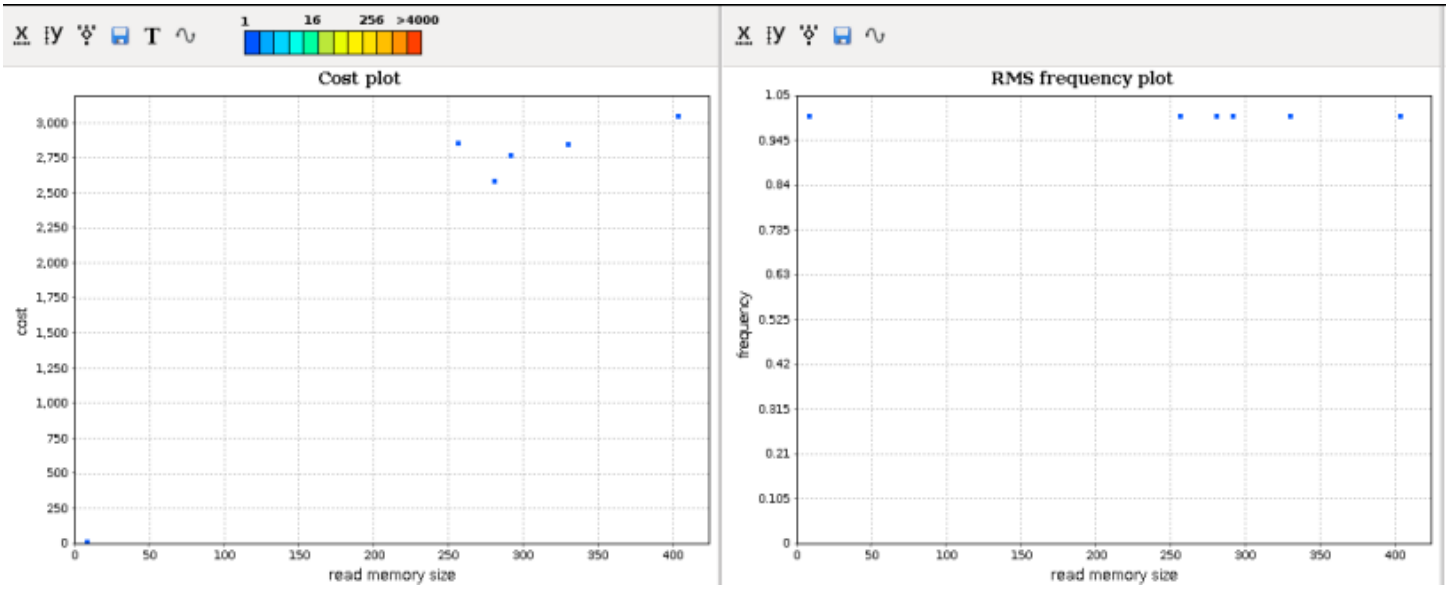
QueueManager__putDownload



BufferedSocket__checkEvents



FinishedManager__onComplete



Observations:

We observed similar pattern that we observed in phase-1. As we go down the list of methods in decreasing order, the input size on which they are invoked and their execution time both are decreasing.

PHASE 3 & 4

In phase -3 & 4, we have instrumented the AUT with different inputs and different input sizes. For each combination, we collected the methods which are accessed from multiple threads. One such combination is downloading multiple files.

When analyzed the set of methods generated for each run, we found there are some common methods that were executed when we fixed the input I.e downloading multiple files and increased the input size.

1) Downloading 6 files
2) Downloading 9 files
3) Downloading 19 files
4) Downloading 50 files

Challenges Faced:

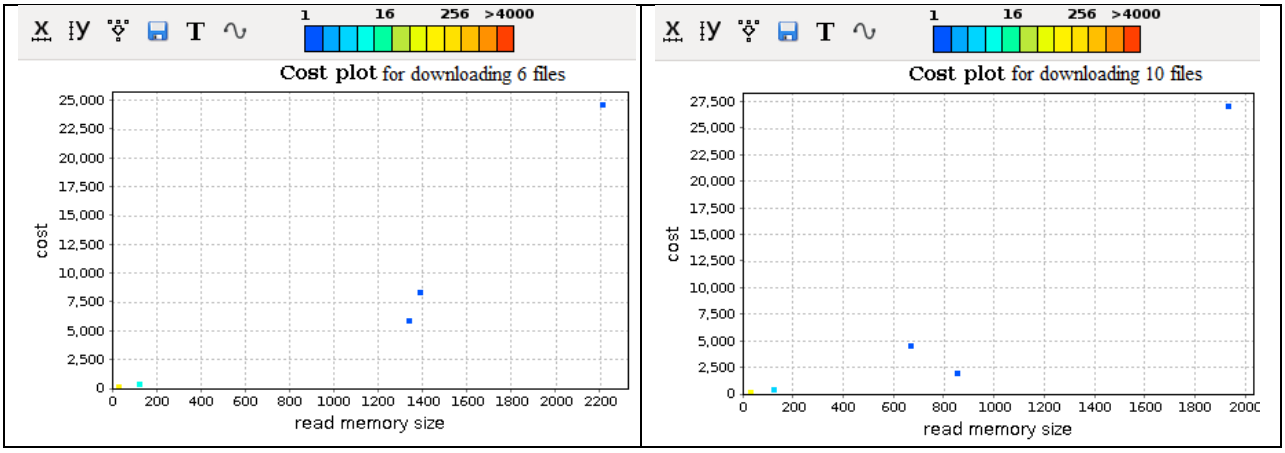
After collecting the methods which are common and also having threads using our automated code, a big roadblock was that some of these methods although listed in report files generated by aprof, they are not showing up in aplot. Hence we are not able to provide plots for such methods. Hence, we have to try more combination of inputs so that we will get some methods, for which we get plots. That is one of the reason why we took "Downloading Multiple Files" as the input for which increased the size.

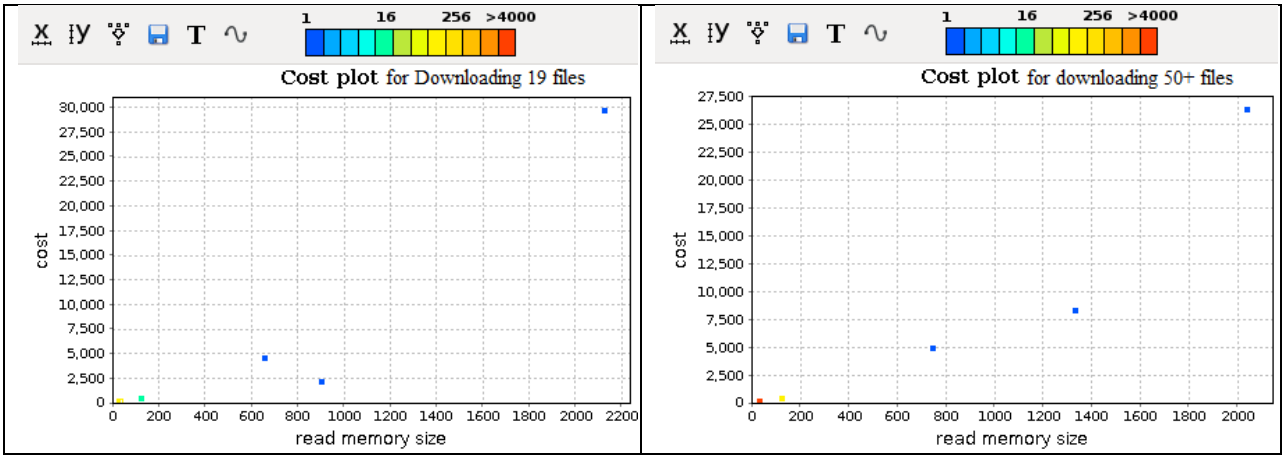
Those common methods were listed below and their plots were analyzed.

Method 1:- BufferedSocket_checkEvents

Input Size	Method Name	Cost
Downloading_50+_files	BufferedSocket__checkEvents	5,495,845
Downloading_multiple_files_10	BufferedSocket__checkEvents	58,721
Downloading_multiple_files_19	BufferedSocket__checkEvents	54,570
Downloading_multiple_files_6	BufferedSocket__checkEvents	50,647

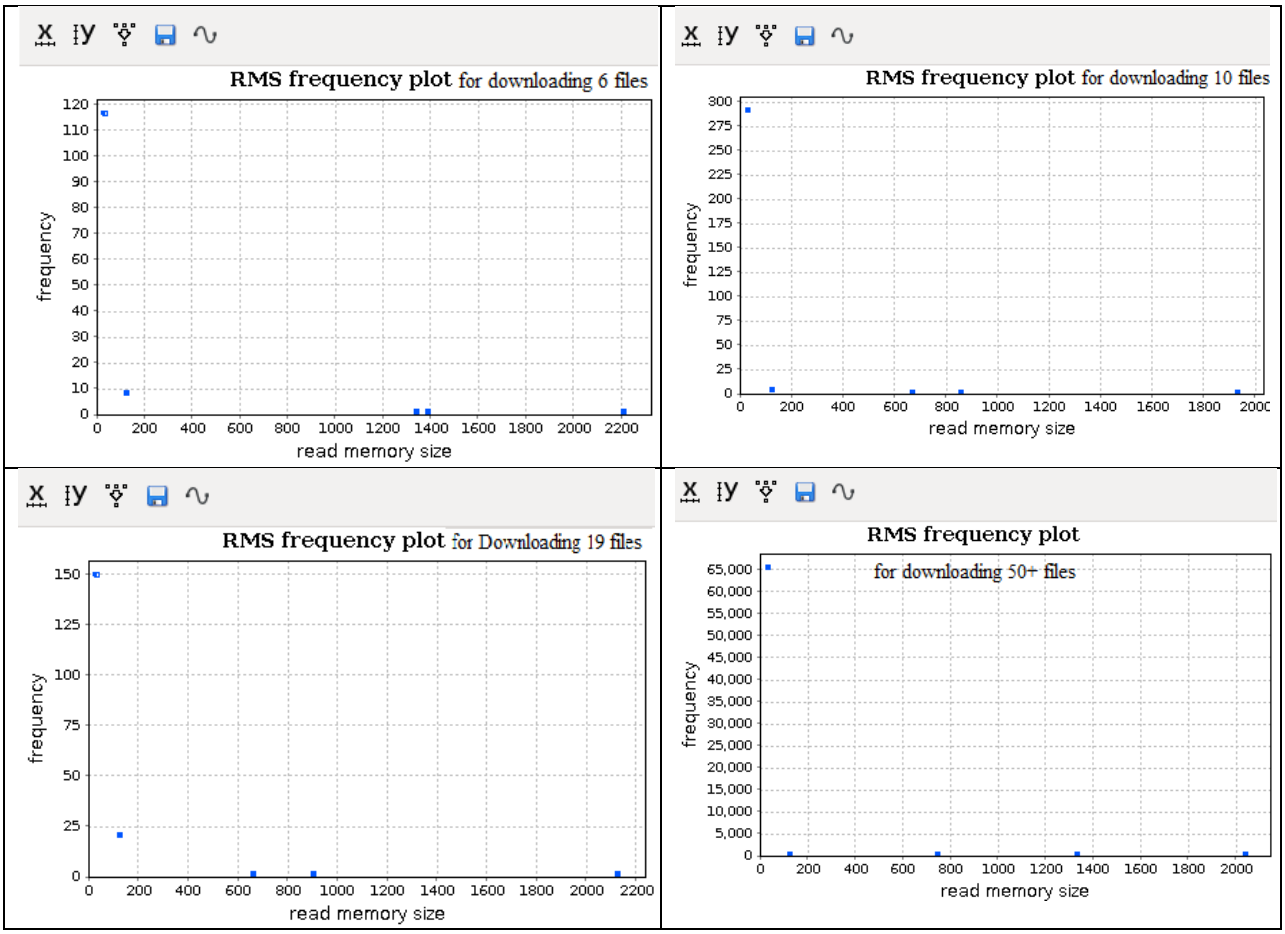
Cost Plot for BufferedSocket_checkEvents :





Observation:
The execution cost increases linearly with input size.

RMS Frequency Plots for ---------- Method:

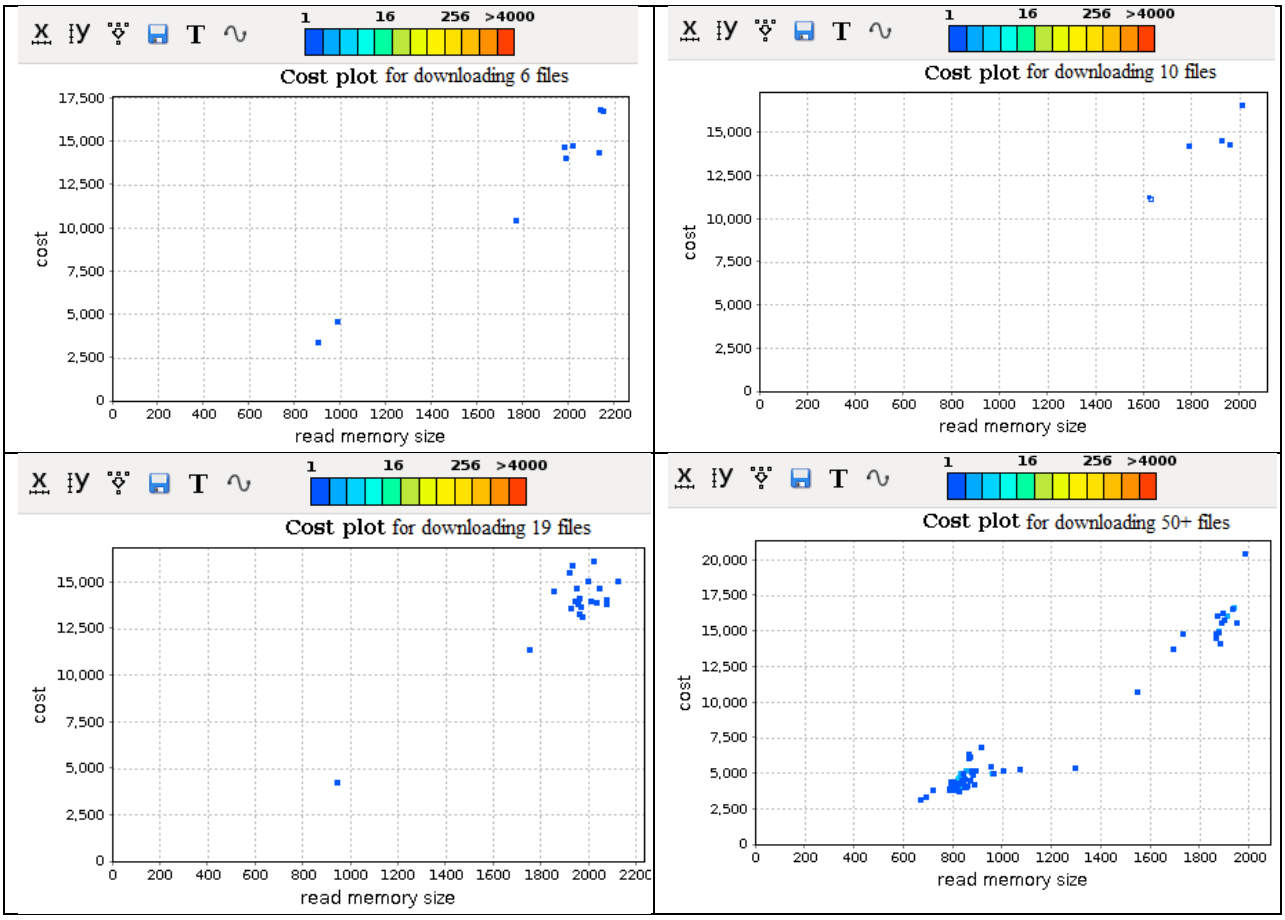


Observation:
Like in the quicksort example, the function is invoked more number of time for smaller input size

Method 2: BufferredSocket__threadRead

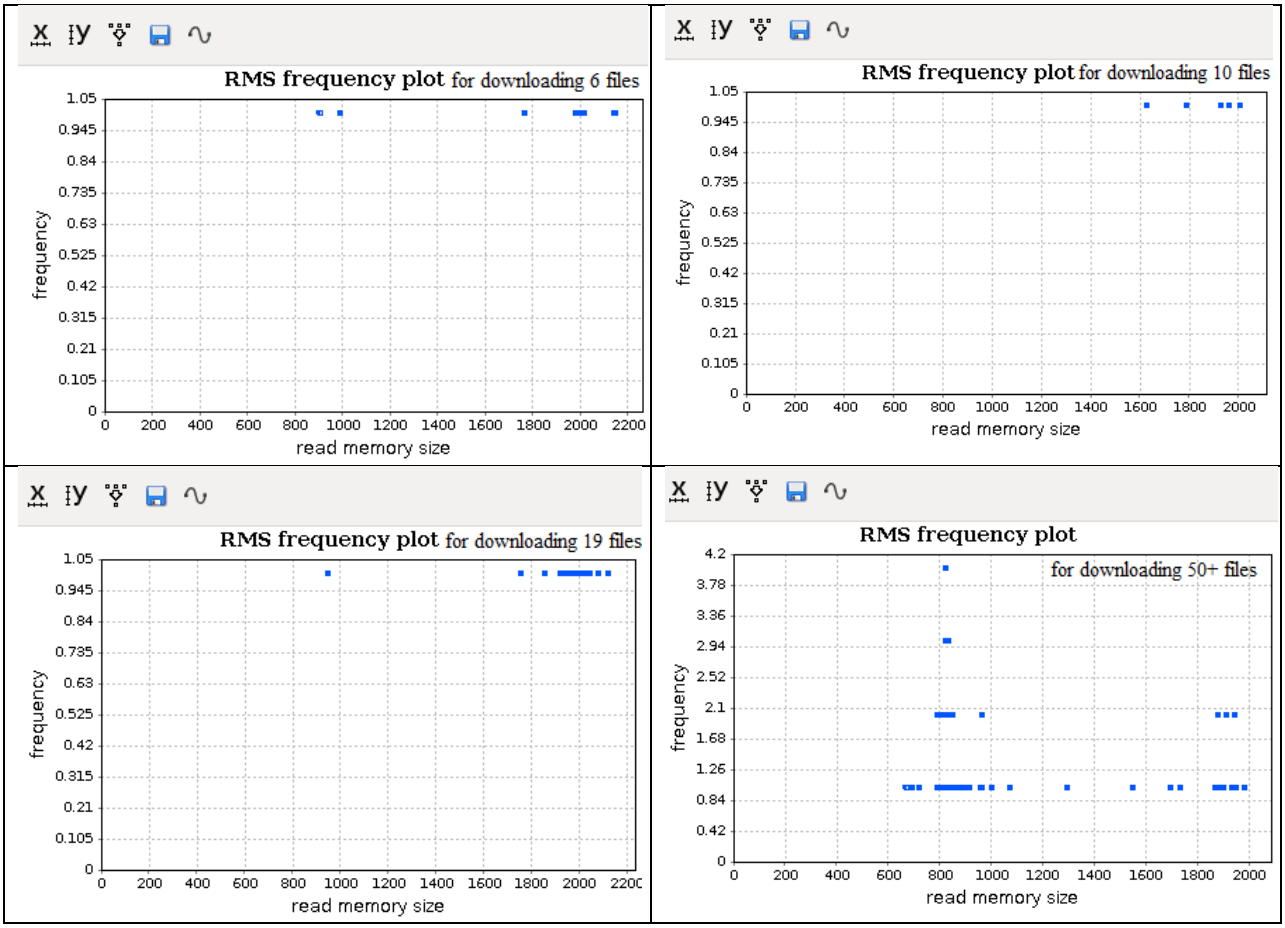
Input Size	Method Name	Cost
Downloading_50+_files	BufferedSocket__threadRead	157,059,642
Downloading_multiple_files_19	BufferedSocket__threadRead	2,846,119
Downloading_multiple_files_6	BufferedSocket__threadRead	1,406,435
Downloading_multiple_files_10	BufferedSocket__threadRead	430,842

Cost Plots for BufferredSocket__threadRead



Observation:
The execution cost is not increasing with input size. It is remaining constant for input sizes.

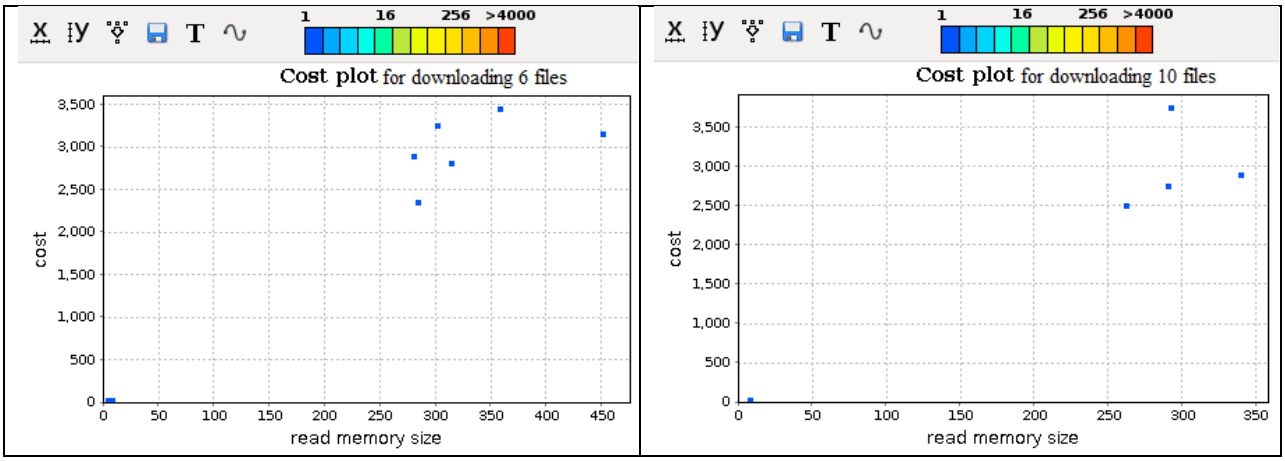
RMS Frequency Plots for BufferredSocket__threadRead:

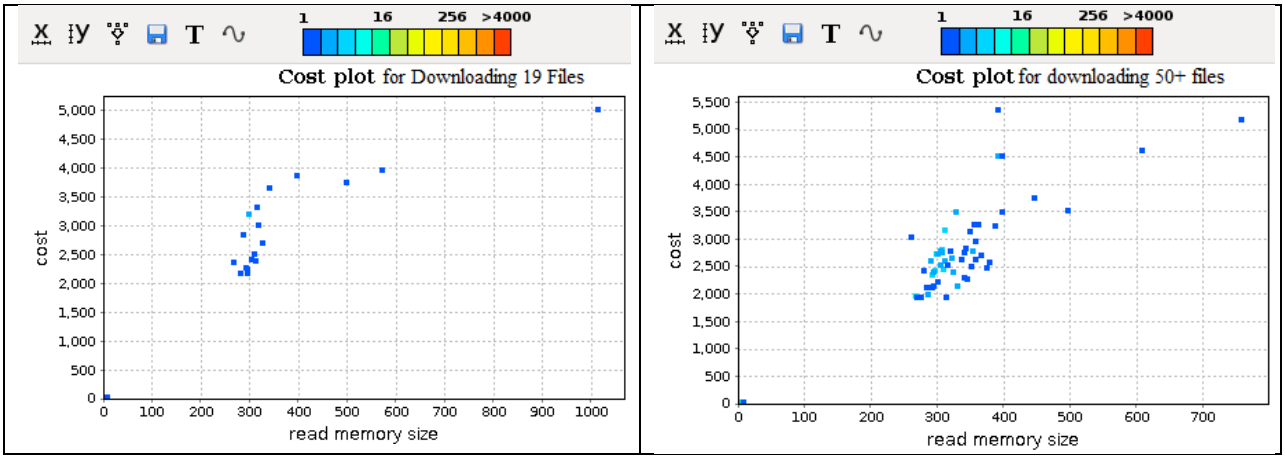


Method 3: FinishedManager__onComplete

Input Size	Method Name	Cost
Downloading_50+_files	FinishedManager__onComplete	223,672
Downloading_multiple_files_19	FinishedManager__onComplete	56,360
Downloading_multiple_files_6	FinishedManager__onComplete	17,844
Downloading_multiple_files_10	FinishedManager__onComplete	11,828

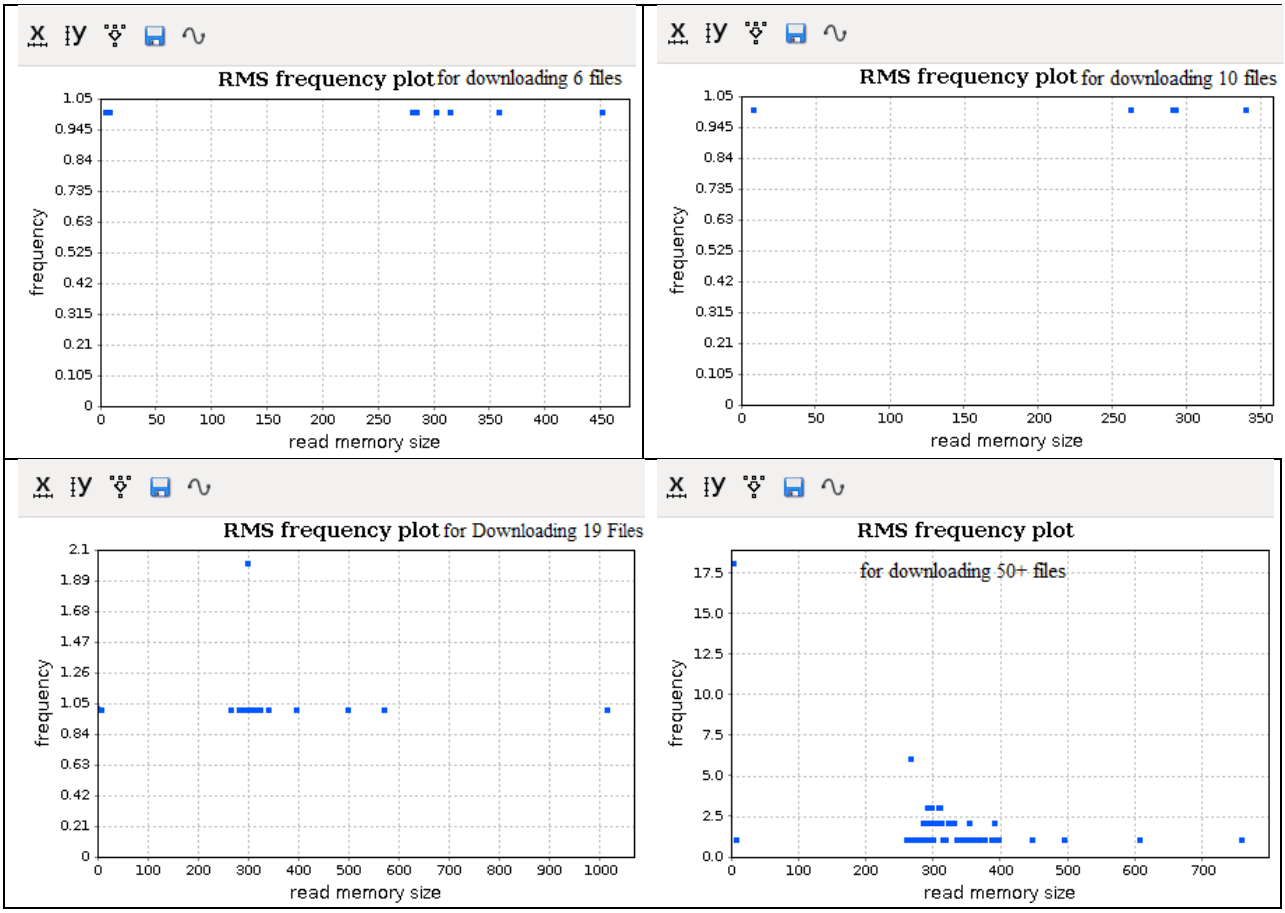
Cost Plots FinishedManager__onComplete:





Observation:
Even though the input size has increased from “Downloading 6 to 50 files”. So the execution cost for the method remains constant for increased input sizes.

RMS Frequency Plots for FinishedManager__onComplete:

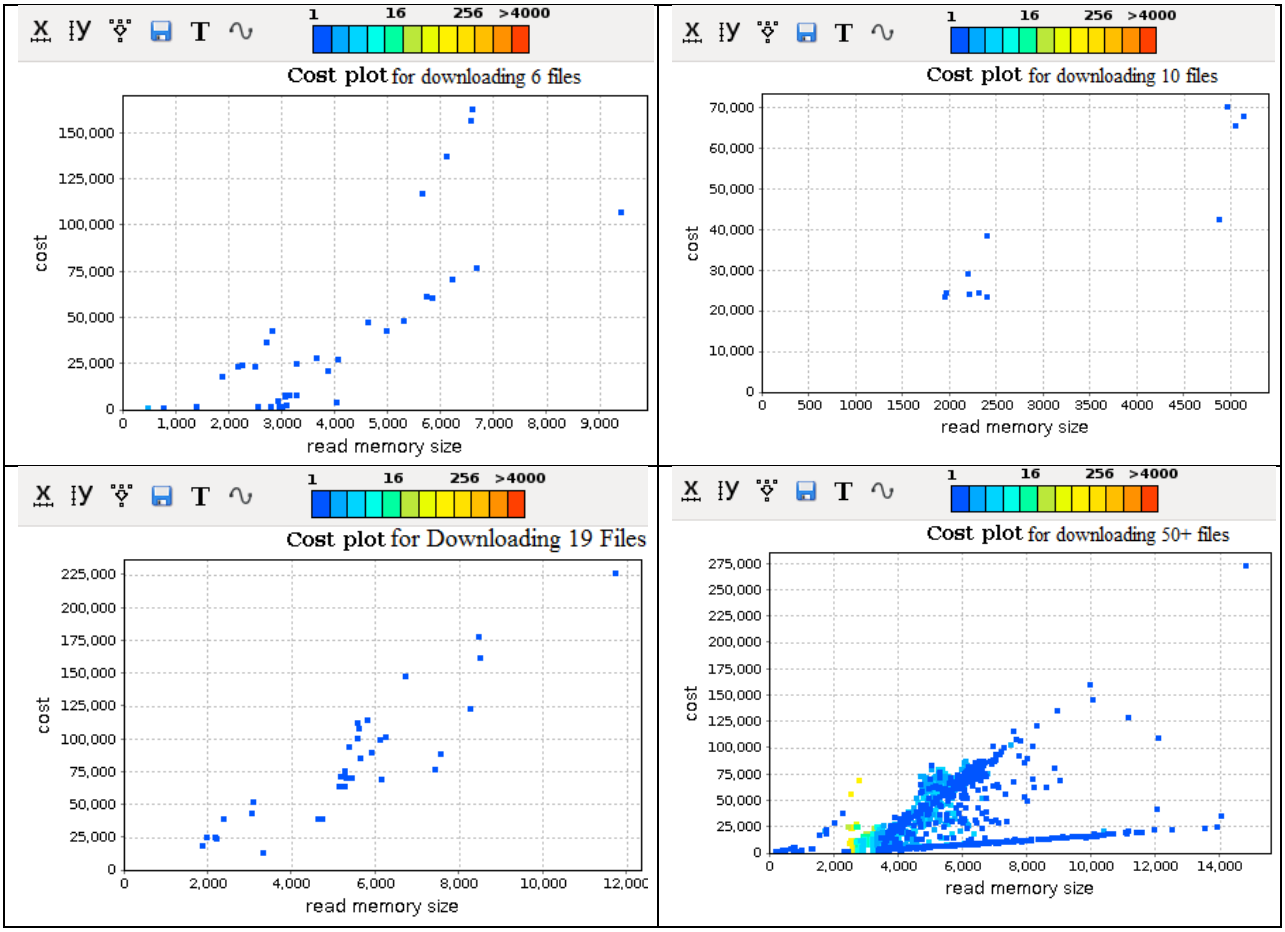


Observation:
For input size of 6, 9 and 19 files, the frequency of method execution Is remaining constant, but for 50 files, the frequency is increased.

Method 4: QueueManager__putDownload

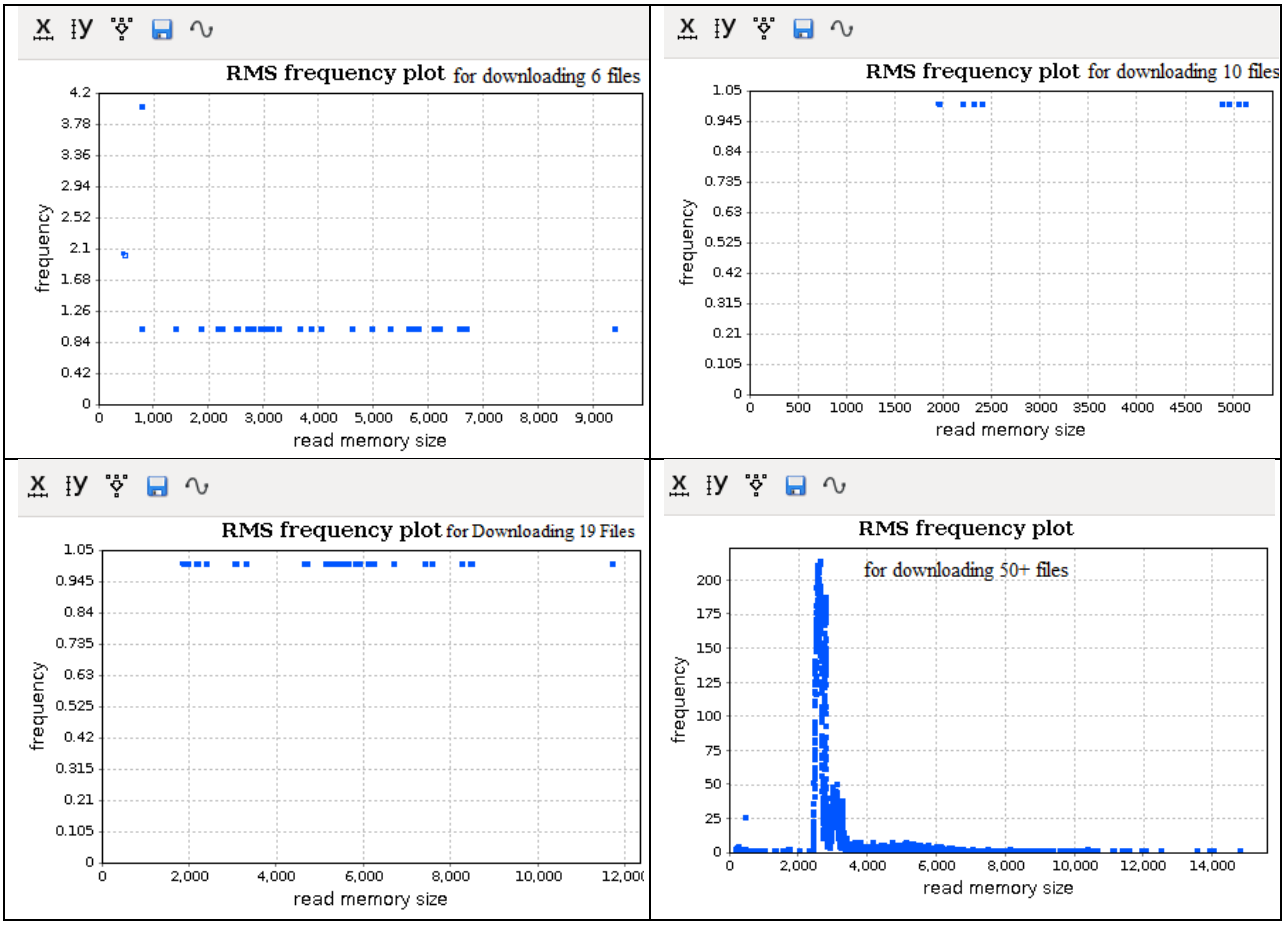
Input Size	Method Name	Cost
Downloading_50+_files	QueueManager__putDownload	668,152
Downloading_multiple_files_19	QueueManager__putDownload	286,896
Downloading_multiple_files_6	QueueManager__putDownload	109,220
Downloading_multiple_files_10	QueueManager__putDownload	70,497

Cost Plots for QueueManager__putDownload:



Observation:
As we increase the input size, the execution cost increases exponentially with input size.
It can be seen in plot for downloading 50+ files.

RMS Frequency Plots for QueueManager__putDownload:



Observation:

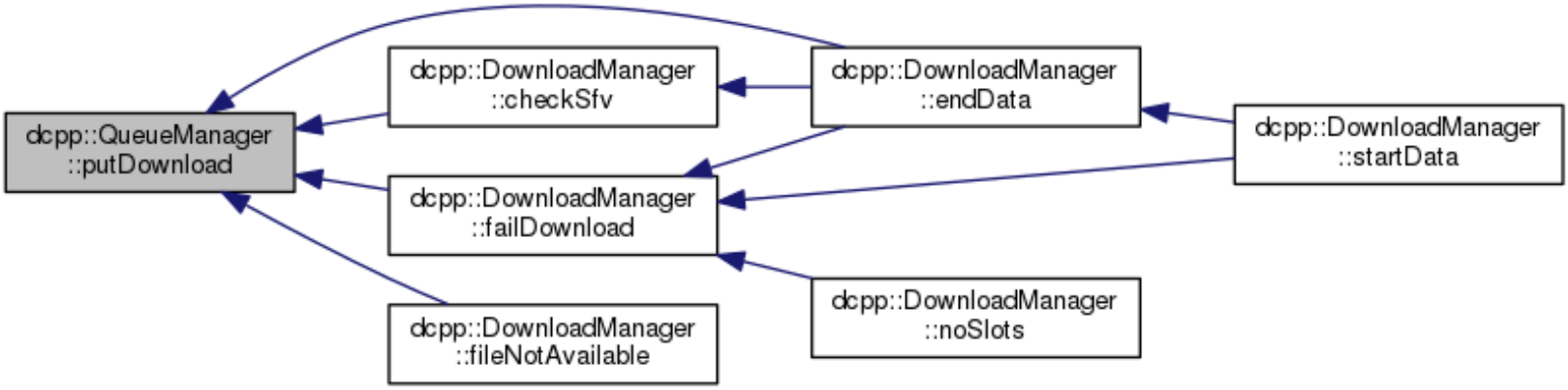
The memory usage and frequency of that method call is increasing exponentially when we are downloading 50+ files.

Conclusion:

The patterns we observed are either increase in execution cost with increase in inputs size or remaining constant for first three methods. But the most important observation is, for *putDownload* as we increase the input size, (in our case, downloading 50+ files) the execution cost varies exponentially. It is also observed that the frequency of that method being called and its memory usage is increasing exponentially. Hence it is a potentially a bottleneck.

Execution Path :

Execution path for the method is analyzed using doxygen.



caller graph for putDownload method

caller graph for endData method

Analysis of code:

From analyzing the code, our understanding is, DownloadManager::startData and DownloadManager::endData are collecting <> for a memory block,
(which is part of file, that is requested to be downloaded) and putting it on a queue to download by calling QueueManager::putDownLoad.

Hypothesis:

So the possible explanation for putDownload method's exponential increase of execution time is that, when we download large number of files (in this case 50), they are called from multiple threads, and since locks are present, these threads are waiting for others threads to release locks.

Also one reason could be that when huge number of threads are created in thread queue, there will be lot of swap-in and swap-out operations(As David explained in class), which will increase execution time by huge factor.

The calle graphs of 'startData' and 'endData' are included as separate images as they are too big to be put in report.

Summary:

1) We took an open-source C++ multithreaded application and instrumented it with valgrind profiler.
2) By using our automated code, we narrowed the list of methods provided by report files, to only those methods which are called from threads.
3) After trying different combination of inputs and input size, we found four methods which are common and are being called from threads.
4) Those methods are invoked for inputs of "downloading multiple files".
5) Hence, the type of input was fixed and its size was increased gradually i.e.
    a) Downloading 6 files
    b) Downloading 9 files
    c) c)Downloading 19 files
    d) d)Downloading 50+ files
6) When we analyzed each method's plot for these input sizes, we observed the pattern of increasing execution cost with increasing input size.
7) But the most import observation is that for the method <> as we heavily load the application by downloading 50+ files, the cost of execution is increasing almost exponentially.
8) It was observed that for this method, memory too increased exponentially.
9) Hence it could be a potential bottleneck for performance.
10) Then by using doxygen, caller-calle graphs were generated to analyze execution path.

References:

[1] http://dcplusplus.sourceforge.net/

[2] https://en.wikipedia.org/wiki/DC%2B%2B

[3] http://valgrind.org/

[4] https://en.wikipedia.org/wiki/Valgrind

[5] https://github.com/ercoppa/aprof/wiki#research-papers

[6] https://github.com/ercoppa/aprof/wiki/Aprof-Manual

[7] https://piazza.com/class/idszvxemu876i4?cid=95

[8] http://ubuntuforums.org/showthread.php?t=193984 (build)

[9] https://acm.cs.uic.edu/uicwifi-linux (connect to uic wifi)

[10] https://www.openhub.net/p/linuxdcpp/analyses/latest/languages_summary