
# Quick Performance User's Manual

CPE 631

KYLE RAY
KRR0010@UAH.EDU

April 30, 2018

# Contents

Goal of Quick Performance ..... 2
Software Dependencies ..... 2
What is Available in Quick Performance? ..... 3
Topology Tab ..... 3
Perf Stat Tab ..... 4
Performance Group Selection Tab ..... 5
Performance Metric Selection Tab ..... 6
Feature Control Tab ..... 7
CPI Stack Tab ..... 8
Application Output Tab ..... 9
Perfscope ..... 9
How to Use Quick Performance ..... 10
Example Use Case ..... 12
Setup ..... 13
Experimental Process ..... 13
Summary and Results ..... 19
Appendix ..... 20
Perf Event Paranoid Enumeration ..... 20

## Goal of Quick Performance

Quick Performance is meant to be a simple to use performance tool. It uses pre-existing tools, such as LIKWID and PERF, and wraps them in an easy to user interface so that a user can gather, rather quickly, performance data on their application or algorithm without any prior knowledge of instrumentation tools or profilers.

## Software Dependencies

Quick Performance uses a few tools to provide feedback to the user and these tools will need to be pre-installed for Quick Performance to work.

1.) LIKWID (Like I Know What I'm Doing)
    a. "Easy to use yet powerful performance tools for the GNU Linux operating system"
    b. Releases can be found at: https://github.com/RRZE-HPC/likwid/releases
    c. Follow the installation instructions found on the site
    d. Gnuplot and feedGnuPlot may need to be installed if not packaged with LIKWID, needed for perfscope.
2.) PERF
    a. Can be installed via Linux package manager (part of linux-common-tools)
3.) Python
    a. Standard on Linux should be pre-installed, if not it will reside in the distributions repository

# What is Available in Quick Performance?

## TOPOLOGY TAB

Upon opening the Quick Performance application, you will be greeted with the "Architecture Topology Tab" which is the result of the likwid-topology command ran on startup. The information is displayed in plain text so that you will have a quick representation of the system architecture.

**Tool Utilized: likwid-topology -c -C -g**

The argument -c provides caching information, -C provides clocking information on the processor, and -g is for graphical output. The option -h will give information on any LIKWID command.

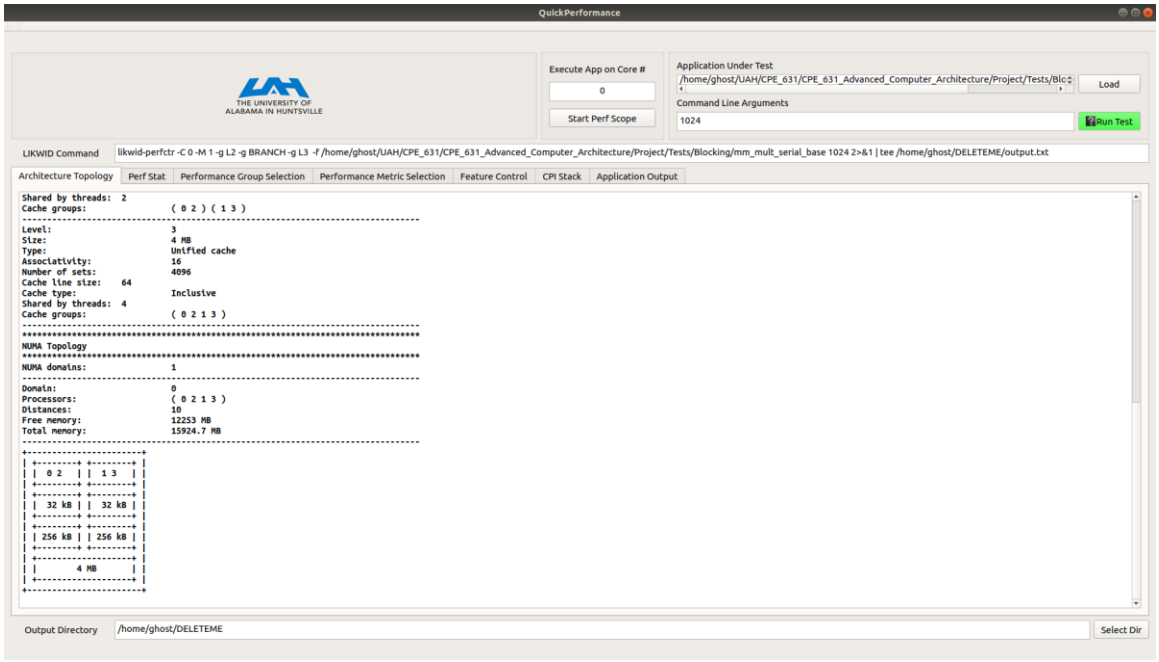


**Figure 1: Quick Performance - Architecture Topology**

This is meant to give a quick overview of the underlying computer architecture to hopefully aid in the software development process if one is seeking to increase performance via software optimization techniques for a certain architecture.

## PERF STAT TAB

The second tab in the list is a stand-alone process, doesn't interact with the other tabs, to give the user immediate feedback on branching, caching, or memory. Allows the user to choose a combination of the above metrics, or choose them all, and proceeds to utilize perf stat to gather information and display the counts as well as create a graphical bar graph to visualize the data. Just make your selections and press the "Run Perf Stat" button to generate results.

**Tool Utilized: likwid-pin -c <coreID> perf stat -C <coreID> -e <metric1, metric2,…> app_under_test**

The likwid-pin command is used to actually pin the application to the specified core so then it would be correct to monitor the application on that core with perf stat. The arguments -c and -C will pin application to a core using LIKWID and monitor a core with perf respectively. The -e will accept a comma separate list of events to monitor and application under test is your application loaded into Quick Performance for analysis.

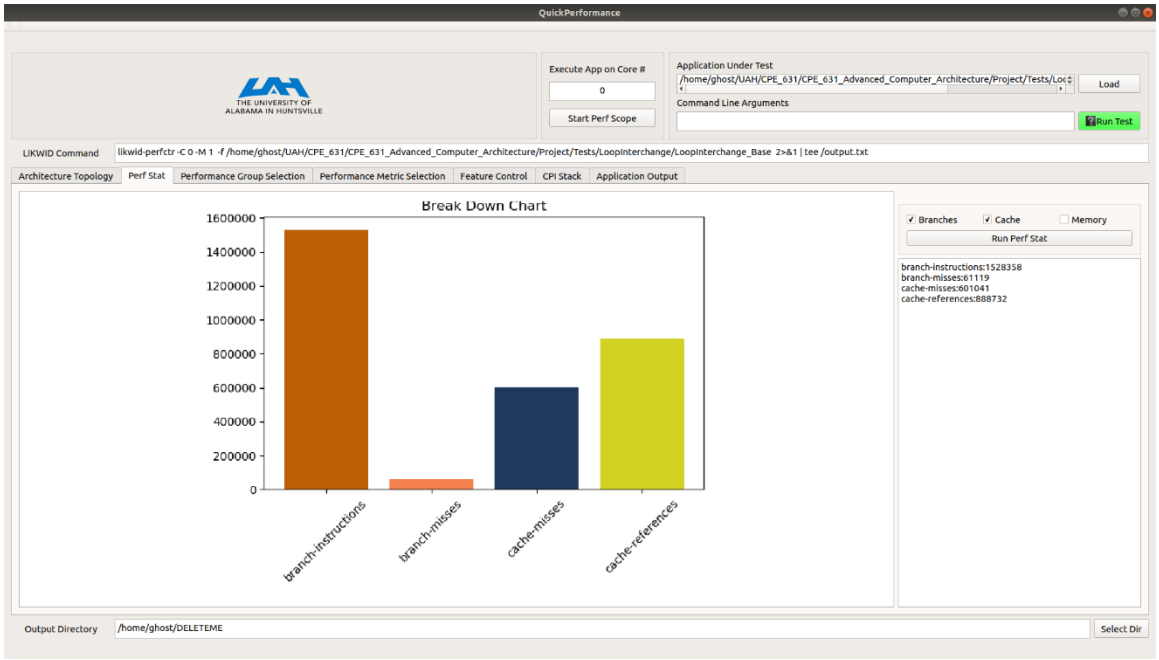


**Figure 2: Quick Performance - Perf Stat Tab**

Note: If you do not see results from pressing the "Run Perf Stat" button then try the following.

1.) Run Quick Performance from the terminal to allow for error information from stderror
2.) Check that perf is installed on the current machine
3.) Modify the perf_event_paranoid file
    a. sudo sh -c 'echo 0 >/proc/sys/kernel/perf_event_paranoid'
        i. Check the appendix for enumeration values for perf_event_paranoid
    b. This will update the paranoid file to allow for CPU monitoring
    c. To make the change permanent you can update for boot settings to do this on startup.

## PERFORMANCE GROUP SELECTION TAB

The Performance Group Selection Tab and the following tab, Performance Metric Selection, allow the user to select from pre-defined scripts to measure certain metrics or the ability to choose from a list of supported events. Double clicking a metric will add it to the chosen performance groups list. If more than one performance group is selected, LIKWID will perform a round-robin method of multiplexing the groups and when they are counted. While it is possible to test with more than one performance group at a time, caution should be used because of the round-robin multiplexing short-running measurements can carry large statistical errors.

**Tool Utilized: likwid-perfctr -C <coreID> -M 1 -g <perfGroup1> -g <perfGroup2> … -f app_under_test**

This tab and the next both utilize the same tool but offer a different set of event selection, where this tab allows selection from pre-defined grouped metrics and the next will allow for more specific event and counter selection. This LIKWID tool allows monitoring of the Model-Specific-Registers (MSR) via the Linux msr module. You will need to start up the msr module for LIKWID to read from the counters, this can be done by running the following command: ***sudo modprobe msr***.

The argument -C will, like likwid-pin -c, pin the following application to the specified CPU core. The -M option sets the access to the msr registers with 0 being direct and 1 being access by LIKWID's access daemon. Because of security reasons using direct access the only option available through Quick Performance is the access daemon, feel free to copy the command and change the access at your leisure. The rest of the command is specifying groups which are directly based on your selection from the list widgets. Refer to the figure below.

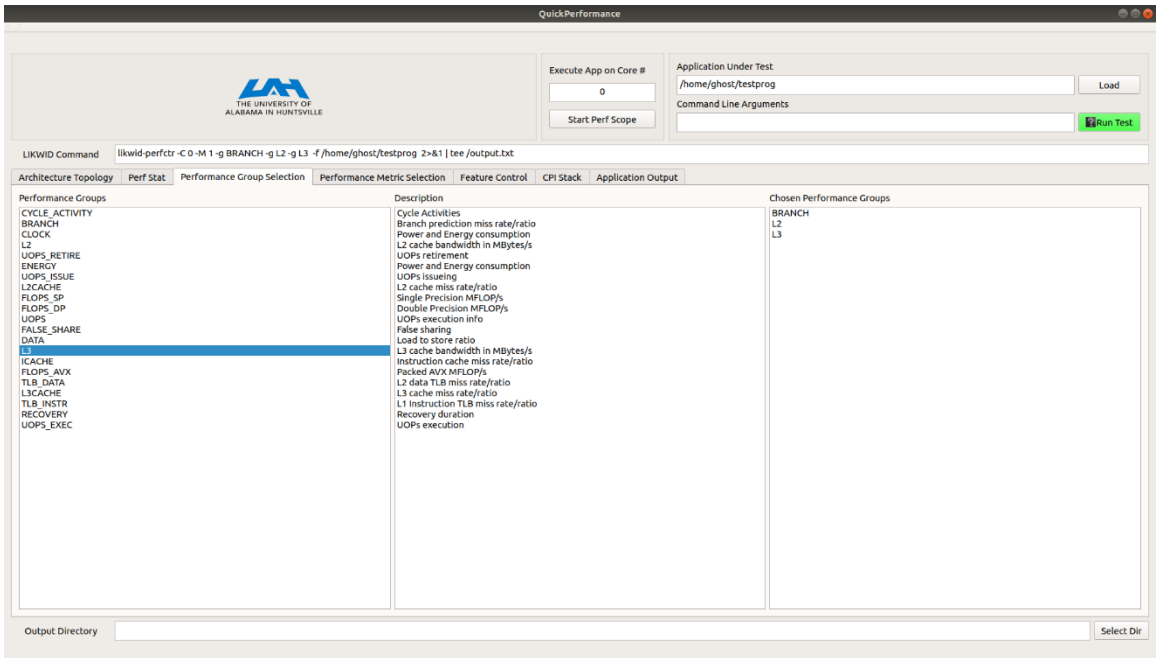


**Figure 3: Quick Performance – Performance Group Selection**

Note: Choosing performance groups will disable the Performance Metrics Tab as Quick Performance cannot guarantee that specific counters aren't already being used by a performance group.

It is possible to create your own performance group. Just follow the examples given in the groups section for your architecture, located in the LIKWID install directory. Once it is in this directory Quick Performance will read it in and you will be able to use it within the application.

## PERFORMANCE METRIC SELECTION TAB

The Performance Metric Selection Tab will allow the user to select from a list of events that are available for their architecture. Double clicking a metric will add it to the chosen performance metrics list. When choosing a metric, the user will have to choose an appropriate counter to monitor that metric. To choose a counter, match the last string in the metric name with the name of a counter. For example, the metric *UOPS_ISSUE_ANY,0xE, 0x1, PMC* will need to be paired with a general performance monitoring counter (PMC) such as *PMC0, Core-local general-purpose counters*. Refer to the figure below for a set of metrics with their appropriate counters.

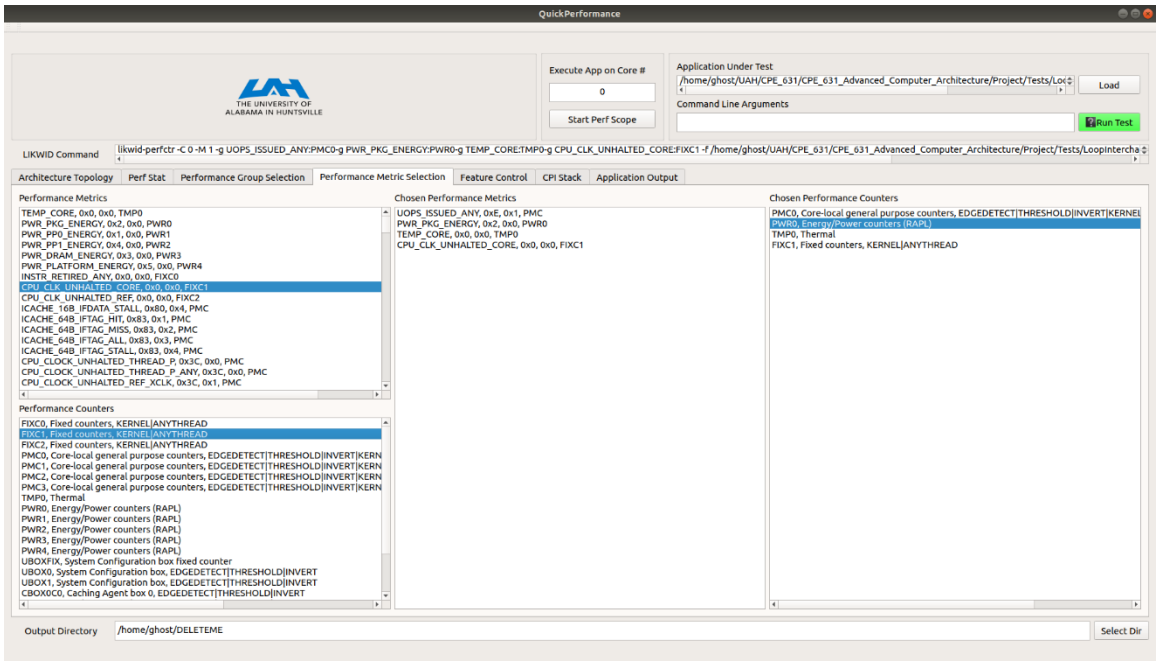


**Figure 4: Quick Performance – Performance Metric Selection with Counter Selection**

Note: It is possible to monitor as many metrics as you have counters available. When choosing performance metrics, the Performance Group Selection Tab will be disabled because Quick Performance cannot guarantee that performance groups won't try and use the same counters selected by the user.

## FEATURE CONTROL TAB

The Feature Control Tab will allow the user to enable/disable certain features that are available on the CPU. To enable/disable a feature just select it from the features list and hit either of the buttons Enable or Disable to toggle the feature, the results of the operation will be displayed in the Operation Results text box. This can be used in conjunction with the "Execute App on Core #" edit box, changing the number value in this box will pull up the features for that CPU core.

**Command Utilized: likwid-features -c <coreID> <-e or -d> <feature>**

This command will either enable (-e) or disable (-d) the selected feature on the core specified with the -c argument.

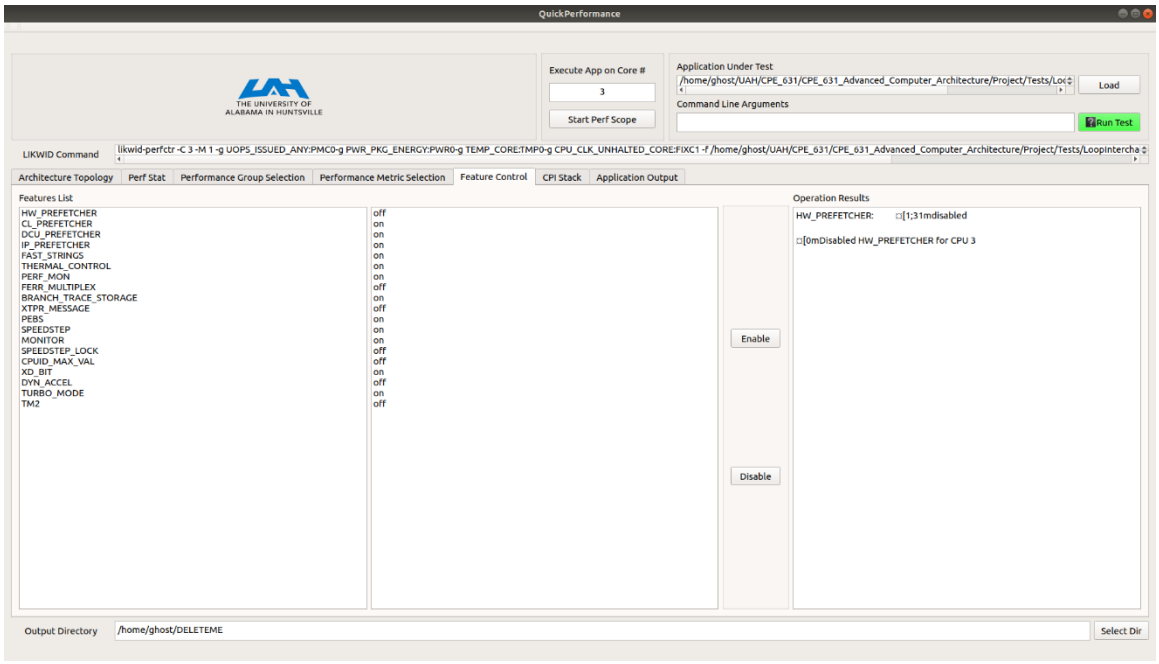


**Figure 5: Quick Performance – Feature Control**

In the image above, we have elected to turn off the hardware pre-fetcher for the CPU core number 3.

## CPI STACK TAB

The CPI Stack Tab is used to show the overall CPI of a performance group. An image will be generated after the user elects to "Run Test" with their current selections from the other tabs. This will run the LIKWID command and pass the data to a python script which in turn will process it into a stacked bar graph displaying the CPI for each performance group category.

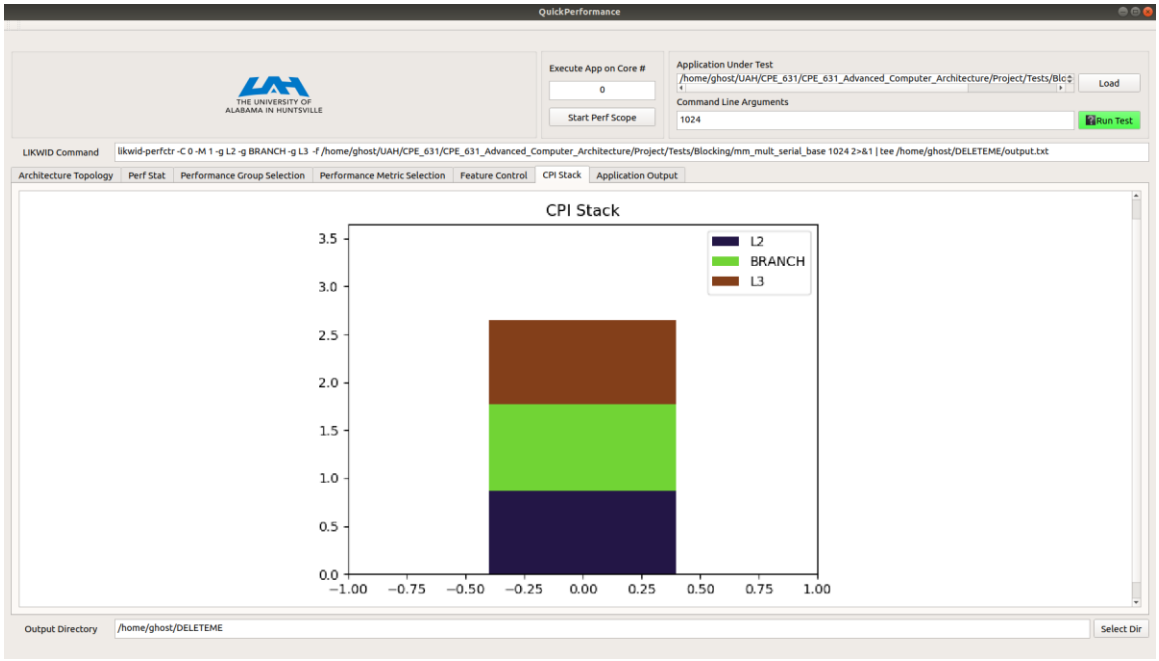


**Figure 6: Quick Performance – CPI Stack**

**WARNING:** Because of my recent findings within LIKWID, the intended implementation of this tab does not function correctly and will possibly be removed if no solution is found to correct the error. Currently displays the overall CPI for the program and not for each individual performance group as originally intended, i.e. for this example each block is approximately the same CPI because it is counting overall application CPI and not each individual component contribution to the overall CPI. Each block represents approximately 0.9 CPI which is the overall CPI for the application in this example, not 2.7 CPI.

## APPLICATION OUTPUT TAB

The Application Output Tab will automatically pull up upon completion of the test and will display the results of the test. The contents of this tab are directly dependent on the choices that are made in the other selection tabs. The header will include some CPU information followed by the output of your application and the counter results. All data will be saved to the selected output directory as "output.txt".

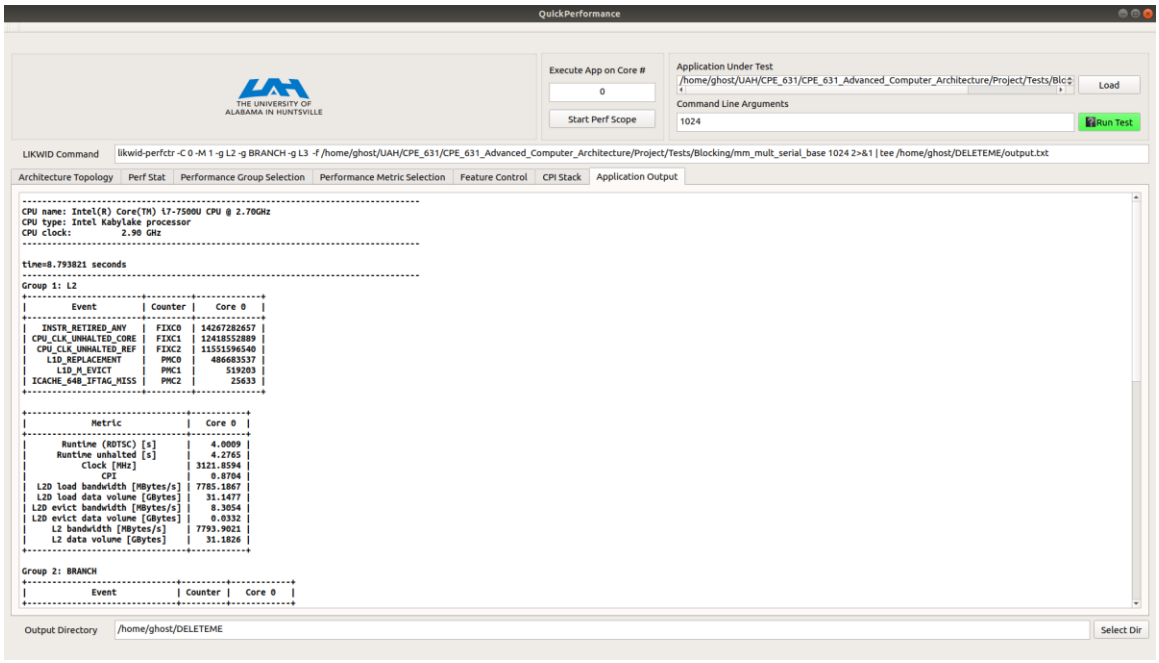


**Figure 7: Quick Performance – Application Output**

## PERFSCOPE

This feature is currently being integrated and developed and is not guaranteed to work. Pressing the "Start Perf Scope" button will trigger the LIKWID perfscope to pull up gnuplots and plot power and energy consumed for the current CPU in real time.

## How to Use Quick Performance

Quick Performance is compiled into an AppImage, which contains all the necessary libraries to run the application.

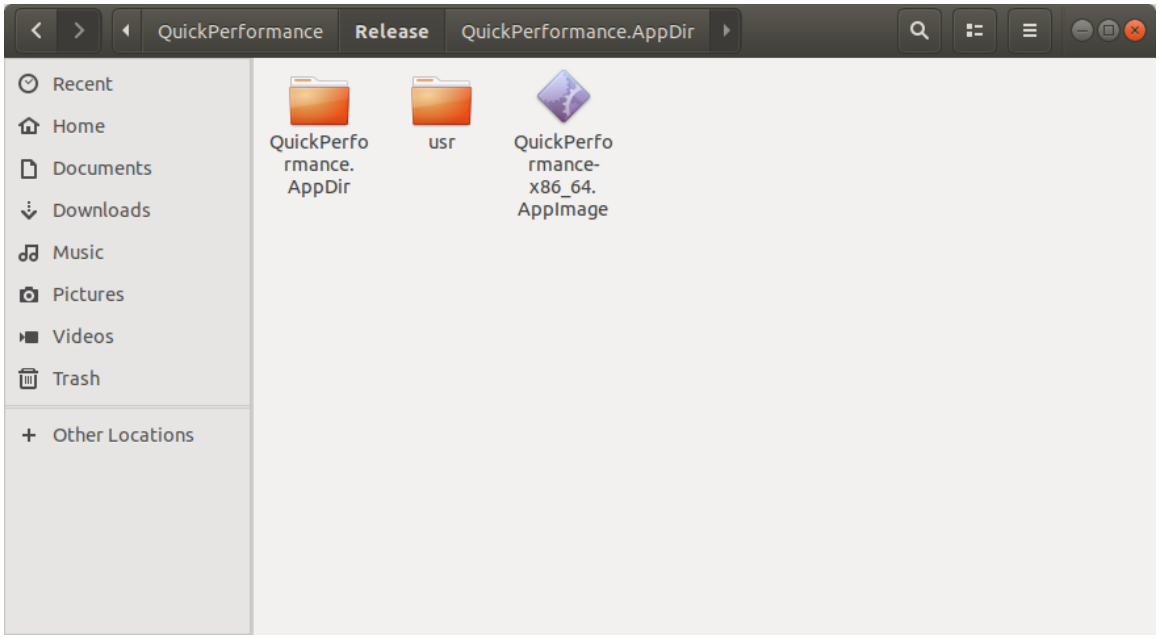


**Figure 8: Quick Performance AppImage and Supporting Folder Layout**

To run Quick Performance simply call the AppImage from the command line, preferred method if you would like to see stdout and stderr output, or double click the AppImage to launch the application. Note that the usr folder contains necessary scripts for Quick Performance to function correctly so make sure to keep this folder with the AppImage. Upon launch you will be greeted with the Topology Tab, see above for description. From here you can elect to just make a quick run using the Perf Stat tab or to move on to the LIKWID tabs and select from performance groups or more specific metrics. While interacting with Quick Performance you may notice that the LIKWID command will update based on your selections and be displayed in the LIKWID COMMAND text box. This is placed here so that it is possible to copy out and be used externally if desired, note that editing of the command inline is not supported.

Once you have made your selections you will need to load your application executable by using the "Load" button. **It should be noted that there is can be a bug with the AppImage for some distributions of Linux where the File Dialog doesn't show any files when trying to load. You can type the absolute path of your file in the dialog if this happens or execute Quick Performance from the AppDir directory by running the AppRun script which should use the native File Dialog.** If your application requires any command line arguments to be run please provide them in the Command Line Arguments text box, refer to the figure below.

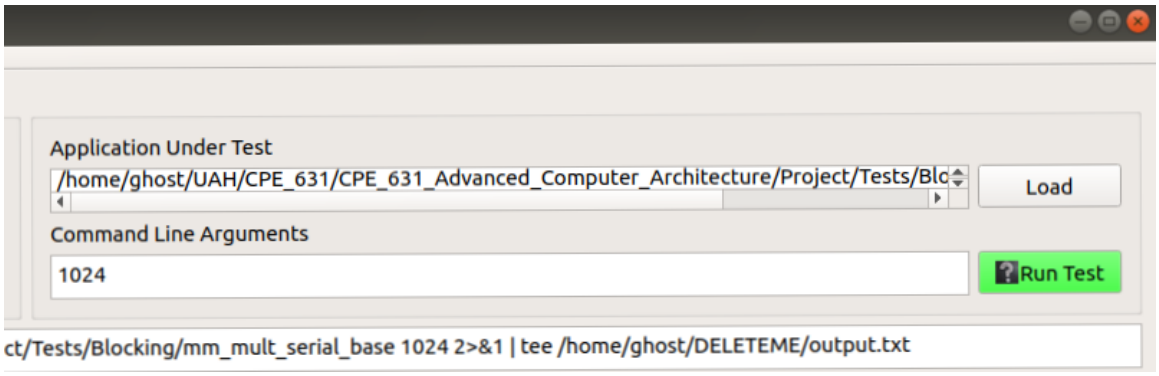


**Figure 9: Load Application and Command Line Arguments**

The "Execute App on Core #" text box will allow you to select the CPU core to pin, execute, and monitor your application on.

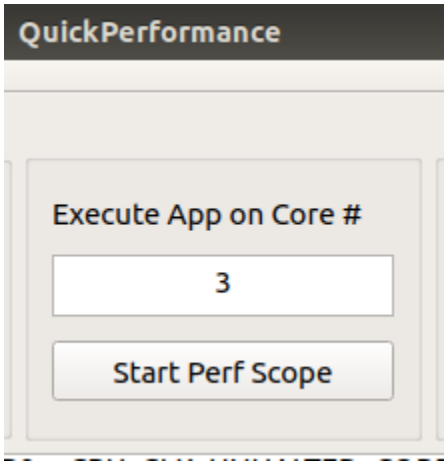


**Figure 10: Core Selection**

It should be noted that the current implementation of Quick Performance does not support multi-threaded applications. If you are interested in changing features of the CPU cores, you can do so before running a test via the Feature Control Tab. Modifying the above Core # will bring up the features for the current core selected.

Once everything is selected then all that is left to do is to press the Run Test button. This will execute the the loaded application under the generated LIKWID command. When the test is finished Quick Performance will automatically pull up the Application Output tab where the output from the loaded application will be displayed and underneath will contain all of data collected by LIKWID

## Example Use Case

In this example use case of Quick Performance we will use the application to gather information on a very simple piece of code, simple multiplication embedded in two loops, and then perform a simple optimization technique, loop interchange, on that code and re-run it through Quick Performance to see if the change was impactful.

Below is the code sample used in this experiment:

```
#include <iostream>
#include <iomanip>
#include <sys/time.h>
using namespace std;

static const int FIRST_DIM = 5000;
static const int SECOND_DIM = 100;

/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((tv2.tv_usec-tv1.tv_usec)+((tv2.tv_sec-tv1.tv_sec)*1000000))
#define TIMER_STOP      gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;


static int x[FIRST_DIM][SECOND_DIM];

void init()
{
        for (int j = 0; j < SECOND_DIM; j = j + 1)
        {
                for (int i = 0; i < FIRST_DIM; i = i + 1)
                {
                        x[i][j] = i;
                }
        }
}

int main(int argc, char* argv[])
{
        init();

        TIMER_CLEAR;
        TIMER_START;

        for (int j = 0; j < SECOND_DIM; j = j + 1)
        {
                for (int i = 0; i < FIRST_DIM; i = i + 1)
                {
                        x[i][j] = 2 * x[i][j];
                }
        }

        TIMER_STOP;

        cout << "time=" << setprecision(8) <<  TIMER_ELAPSED/1000000.0
        << " seconds" << endl;

        return 0;
}
```

The code just initializes a 5000x100 array of integers and then doubles the value.

## SETUP

The machine setup for this experiment consists of the following hardware:

CPU: Intel core i7-7500U

CPU Clock: 2.90 GHz

CPU Type: Quad Core Kaby Lake Processor

Cache:

- L1: 32 KB 8 way set-associative data cache
- L2: 256 KB 4 way set-associative unified cache
- L3: 4 MB 16 way set-associative unified cache

Memory : 16 GB DDR4

## EXPERIMENTAL PROCESS

First, we need to load in the application and make our selections on what we want to monitor for the application and select an output directory to save the results. For this setup, we are going to be pinning our application onto the second core (core id 1 because of zero-base) and checking the affect on branching and the caches inflicted by our application. Running the perf stat option will give very quick feedback and then if we desire we can probe further with LIKWID.

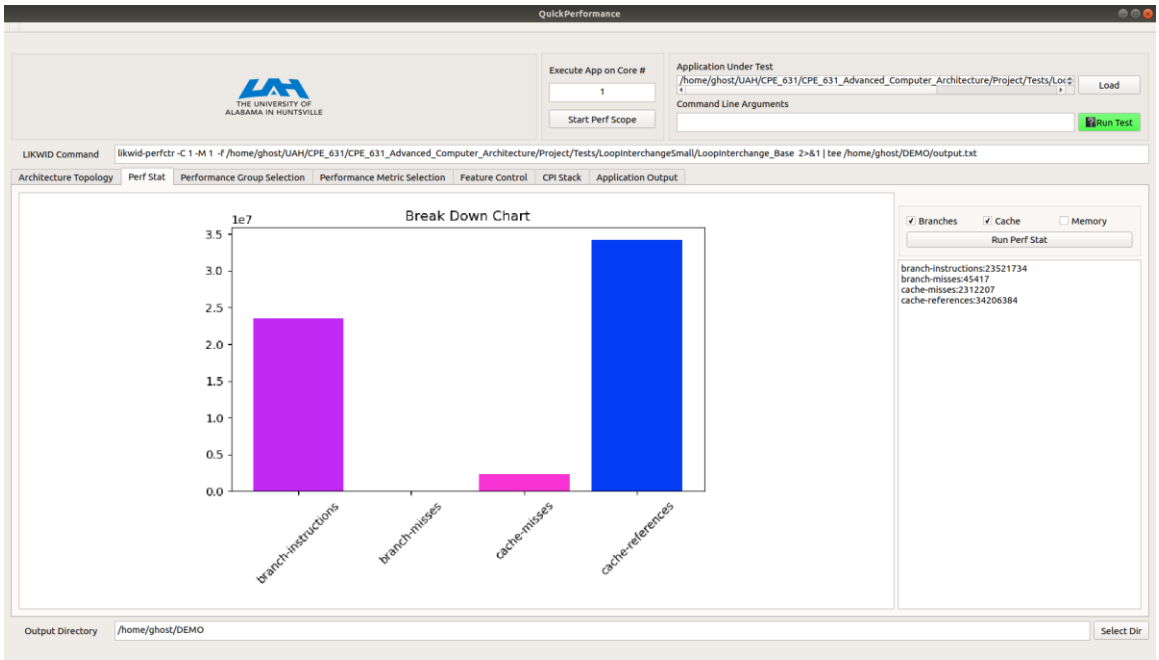


**Figure 11: Current Experiment Setup and Perf Stat Output**

At first glance, we can see that we have approximately 34 million cache references, note that this doesn't delineate which level cache caused these references and we can use LIKWID groups to get more information on the impact at each level. We can also see that we have approximately 2.3 million cache misses, 23.5 million branch-instructions, and 45,000 branch misses.

**NOTE: RUNNING THE BR_INST_RETIRED_ALL_BRANCHES VIA LIKWID WE GOT THE NUMBER 20,360,066 branch instructions which is close to what perf stat reported**

While everything is loaded we can elect to choose the L2 performance group to see the impact our application has on L2 cache. This works by selecting the Performance Group Selection tab and choosing the L2 performance group, then hitting the Run Test button to gather the results. With a quick switch we can also measure L3 cache information. Refer to the figures below.

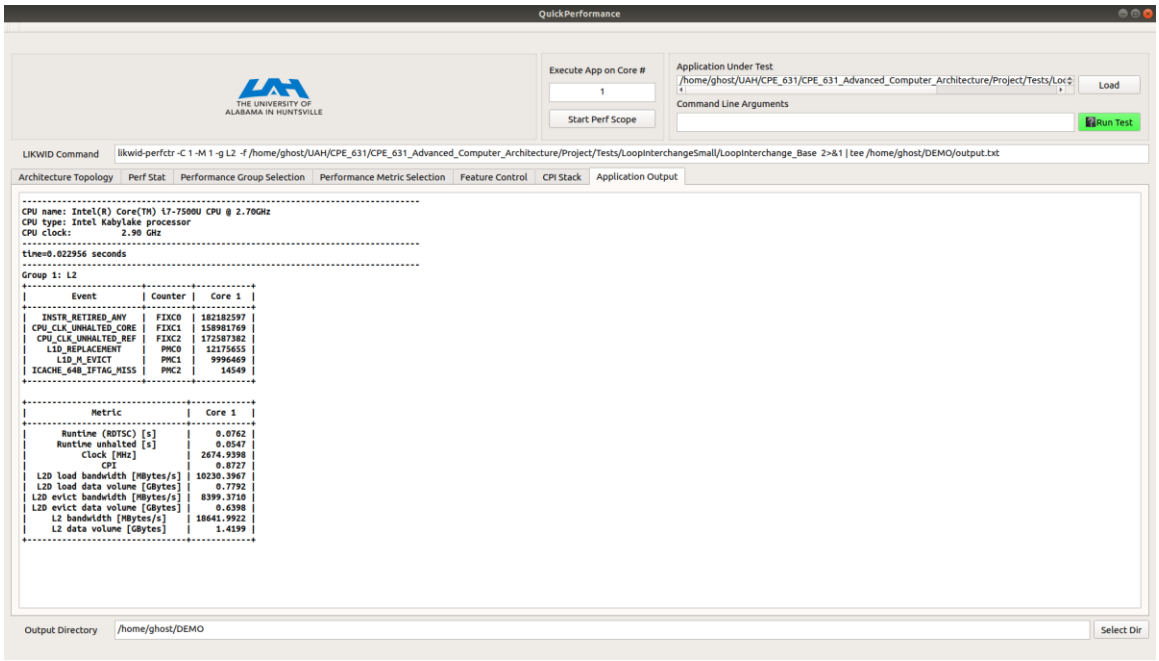


**Figure 12: Experiment L2 Cache Information**

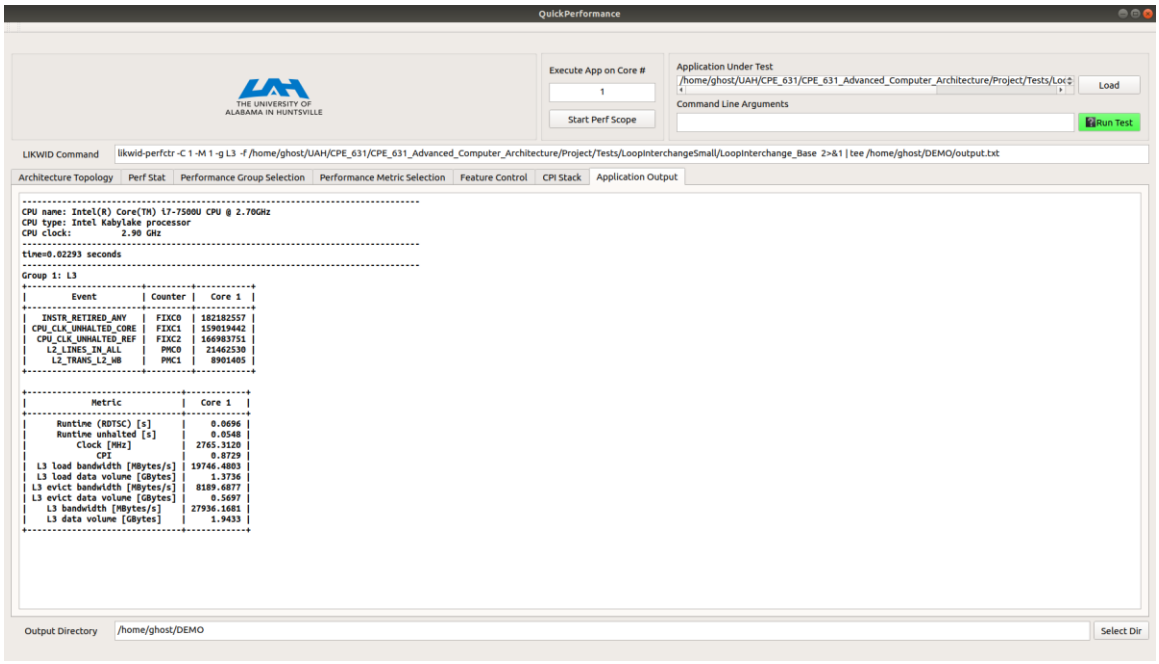


**Figure 13: Experiment L3 Cache Information**

Now that we have a baseline we will perform the simple software optimization technique of loop interchanging. Some applications, our experiment example included, have nested loops that access data in memory in a non-sequential order. Exchanging the nesting of the loops can make the code access the data in the order in which they are stored. This technique reduces misses by improving spatial locality because reordering maximizes use of data in a cache block before they are discarded. Exchanging the nesting of the loops and re-running it through Quick Performance provides the following. For clarification the change in source code has been included below.

```
#include <iostream>
#include <iomanip>
#include <sys/time.h>
using namespace std;

static const int FIRST_DIM = 5000;
static const int SECOND_DIM = 100;

/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((tv2.tv_usec-tv1.tv_usec)+((tv2.tv_sec-tv1.tv_sec)*1000000))
#define TIMER_STOP      gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;


static int x[FIRST_DIM][SECOND_DIM];

void init()
{
        for (int i = 0; i < FIRST_DIM; i = i + 1)
        {
                for (int j = 0; j < SECOND_DIM; j = j + 1)
                {
                        x[i][j] = 0;
                }
        }
}

int main(int argc, char* argv[])
{
        init();

        TIMER_CLEAR;
        TIMER_START;

        for (int i = 0; i < FIRST_DIM; i = i + 1)
        {
                for (int j = 0; j < SECOND_DIM; j = j + 1)
                {
                        x[i][j] = 2 * x[i][j];
                }
        }

        TIMER_STOP;

        cout << "time=" << setprecision(8) <<  TIMER_ELAPSED/1000000.0
        << " seconds" << endl;

        return 0;
}
```

Note: We are following the same steps as before by running Perf Stat first.

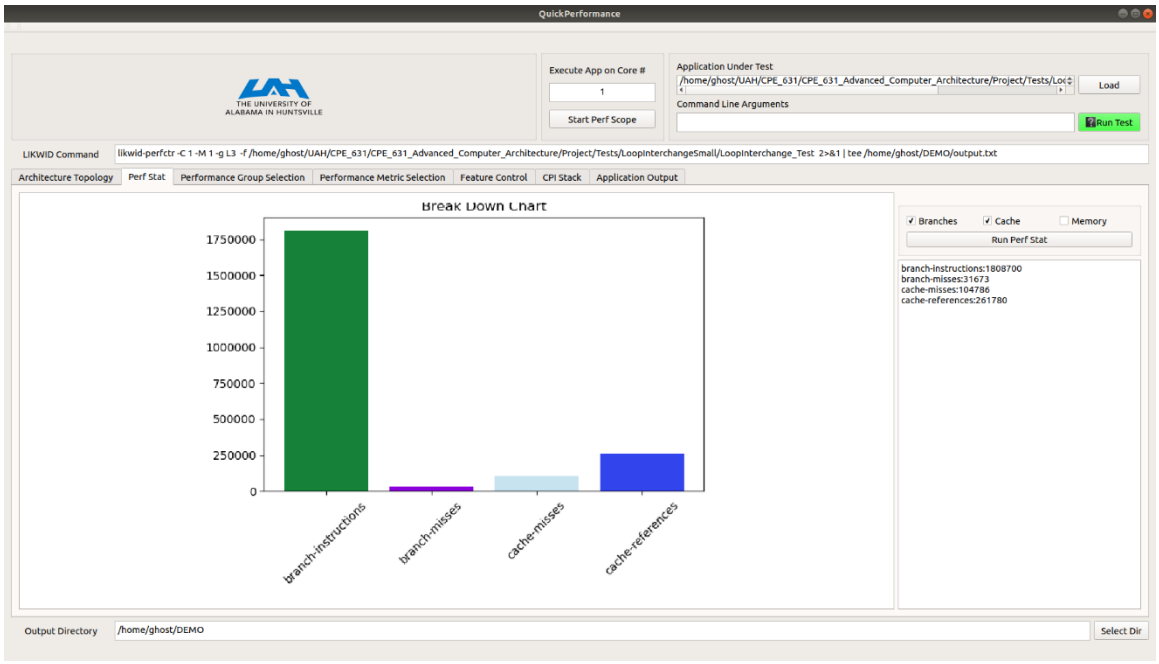


**Figure 14: Optimized Experiment Setup and Perf Stat Output**

We can see now that our cache-references have dropped significantly from approximately 34 million to 261,000. This is a significant decrease in overall references to the cache if we take a quick look at the other metrics we will see a significant decrease as well. Overall with the new implementation there are approximately 104,000 cache misses down from 2.3 million, 18 million branch-instructions down from 23.5 million, and 31,600 branch-misses down from 45,000. Next, we will just re-run with the LIKWID L2 and L3 groups to get the information for the cache levels.

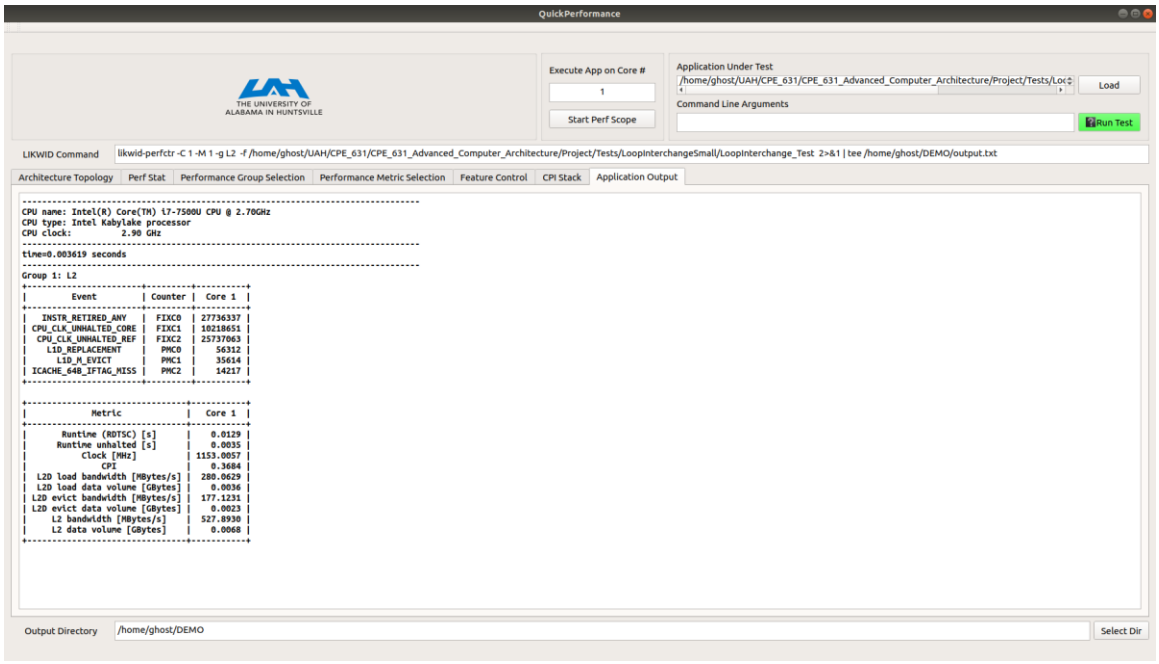


**Figure 15: Optimized Experiment L2 Cache Output**

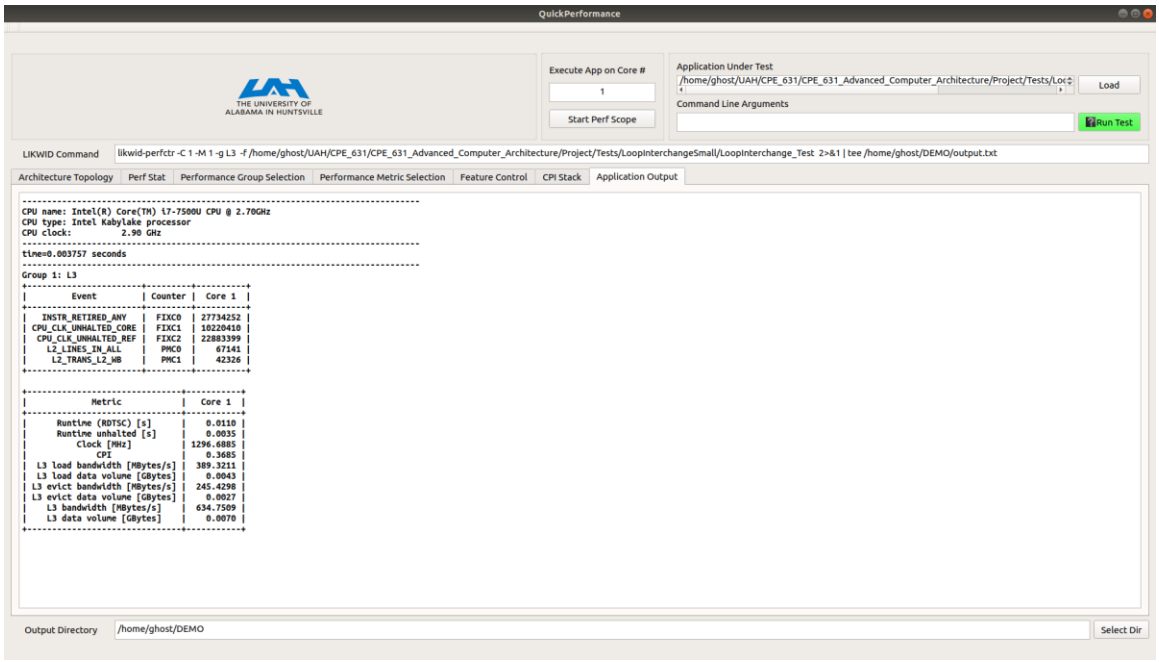


**Figure 16: Optimized Experiment L3 Cache Output**

Summary and tabularized results can be found in the next section

## SUMMARY AND RESULTS

This experiment was meant to demonstrate how easy it is to use Quick Performance, without having to have prior knowledge of instrumentation and profiling tools, to quickly gather performance data on an application. This was a very simple example in that there were only two steps moving from base application and data to the optimized solution and data. Practically there will exists many development cycles in between these two ends of the spectrum and it is my hopes that developers could use this tool to gather data during the process.

**Table 1: Experimental Results**

| | Base | Optimized |
|---|---|---|
| Branch Instructions | 23521734.00 | 1808700.00 |
| Branch Misses | 45417.00 | 31673.00 |
| Cache References | 34206384.00 | 261780.00 |
| Cache Misses | 2312207.00 | 104786.00 |
| L2 Bandwidth (MB/s) | 18641.99 | 527.89 |
| L2 Data Volume (GB) | 1.42 | 0.01 |
| L3 Bandwidth (MB/s) | 27936.17 | 634.75 |
| L3 Data Volume (GB) | 1.94 | 0.01 |
| Time (seconds) | 0.0023820 | 0.0025220 |

For this example, with 5000x100 we don't see a significant speed up in terms of execution time but when running the case with 50000x1000 the base case is 0.744564 seconds and the optimized is 0.109863 seconds which is significant.

In summary, using Quick Performance allows the user to gather this important performance data rapidly so that the user's focus can be spent on development and making their product better rather than spending significant time trying to gather this information with more complex tools. I will state that this is not a replacement for the more complex tools out there as they will generally dig deeper and allow for more insight into how your application is performing, but it is my hope that Quick Performance will be first stop in performance analysis and move on to the more complex and harder to use tools as necessary.

# Appendix

## PERF EVENT PARANOID ENUMERATION

These are the enumerations for the perf_event_paranoid file

-1: Allow use of (almost) all events by all users

>= 0: Disallow raw tracepoint access by users without CAP_IOC_LOCK

>= 1: Disallow CPU event access by users without CAP_SYS_ADMIN

>= 2: Disallow kernel profiling by users without CAP_SYS_ADMIN