Quick Performance Project Report

CPE 631

KYLE RAY

April 30, 2018

Contents

Abstract

A simple performance analysis tool named Quick Performance has been developed to allow software developers an easy way to gather performance information about their applications. The tool utilizes common open source analysis tools such as LIKWID (Like-I-Know-What-I'm-Doing) developed by the HPC group of the computing center at the university of Erlanger-Nuremburg (https://github.com/RRZE-HPC/likwid) and PERF which is a common Linux performance analysis tool.

The Quick Performance tool for this experiment is being used to analyze some simple yet effective software optimization techniques. This experiment will hopefully demonstrate Quick Performance ability to analyze and present meaningful data in a rapid manner. This experiment is a proof of concept for the development process and the role that Quick Performance could play in helping optimize algorithms during or after the development process.

Introduction

In this day and age, we have witnessed the end of Moore's Law. It is no longer the heavenly dream of the software developer to wait on the next great hardware update to provide their applications with out of the box speed ups. It is now up to the software developer to harness/exploit the power of current hardware via software optimization techniques. While there exists a plethora of software profiling and analysis tools to provide the developer with useful information on how to optimize their code, most of these tools have a steep learning curve.

For this project, a simple user interface has been created named Quick Performance that provides the user with some initial analysis information on their application to hopefully facilitate the optimization process. This interface utilizes some popular open source tools, LIKWID and PERF, to gather information about the platform architecture as well as performance information for the user application under test.

The following report contains the information and results where the tool was used to analyze some of the simpler software optimization techniques such as loop interchanging and blocking optimization; Please refer to the User's Manual for Quick Performance on how the tool can be used as this report does not present this detail.

Environment

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

Results

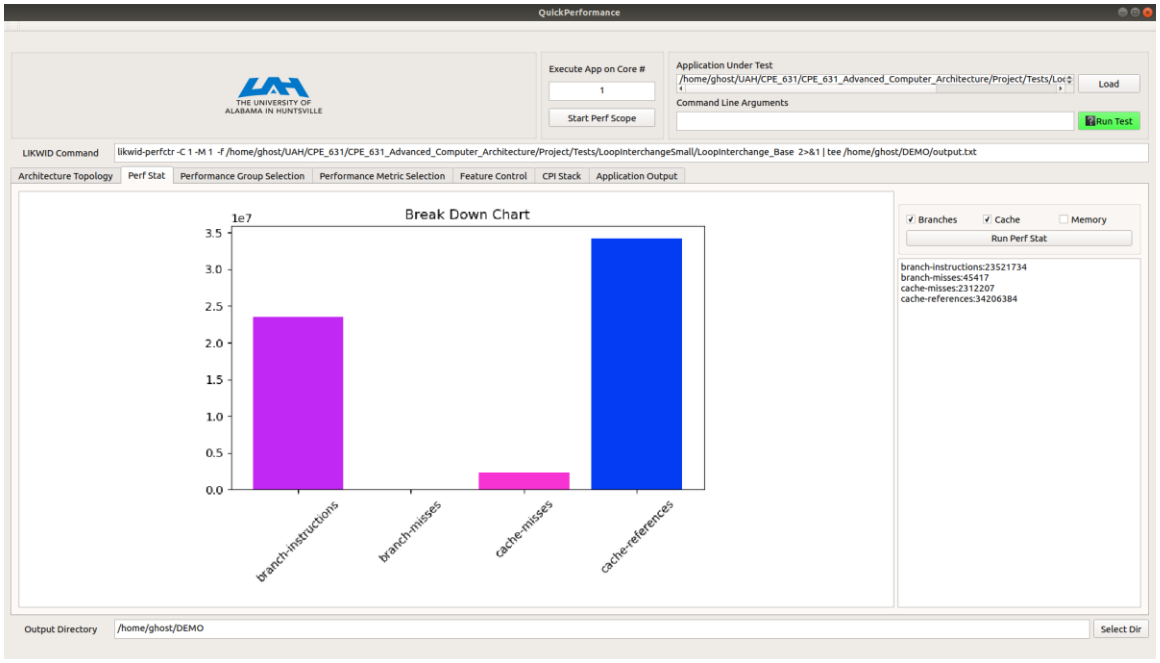
LOOP INTERCHANGE

In this experimental 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.

Refer to the appendix for the source code used in this experiment:

Here is the optimized code, notice that the nested loops have been interchanged which maximizes use of data in a cache block before they are discarded.

Refer to the figures below for the output from Quick Performance when monitoring the example pieces of code above.



Execute App on Core #: 1

Application Under Test: /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/Loc

Command Line Arguments:

LIKWID Command:

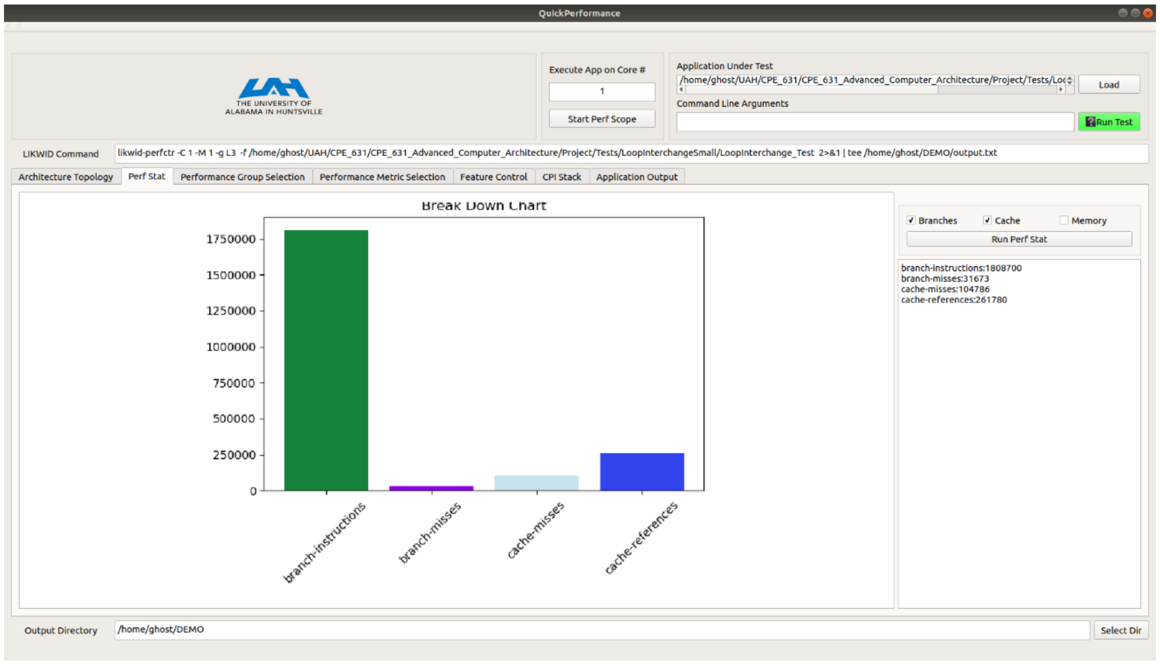
```
likwid-perfctr -C 1 -M 1  -f /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/LoopInterchangeSmall/LoopInterchange_Base  2>&1 | tee /home/ghost/DEMO/output.txt
```

[✓ Branches] [✓ Cache] Memory

```
branch-instructions:23521734
branch-misses:45417
cache-misses:2312207
cache-references:34206384
```

Output Directory: /home/ghost/DEMO

Figure 1: Current Experiment Setup and Perf Stat Output



Application Under Test: /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/Loc

LIKWID Command:

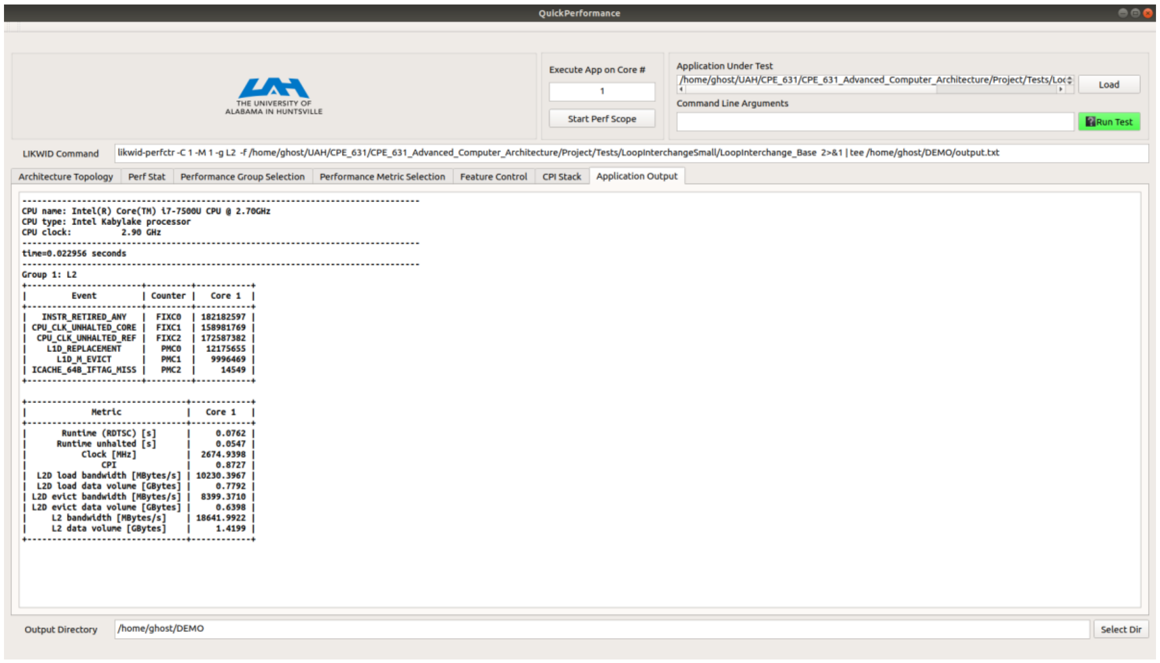
```
likwid-perfctr -C 1 -M 1 -g L3  -f /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/LoopInterchangeSmall/LoopInterchange_Test  2>&1 | tee /home/ghost/DEMO/output.txt
```

```
branch-instructions:1808700
branch-misses:31673
cache-misses:104786
cache-references:261780
```

Output Directory: /home/ghost/DEMO

Figure 2: Optimized Experiment Setup and Perf Stat Output

Using the Perf Stat option in Quick Performance we can compare the executions of the base and optimized solutions and find that the optimized solution is orders of magnitude lower in some of the metrics measured.



Application Under Test: /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/Loc

LIKWID Command:

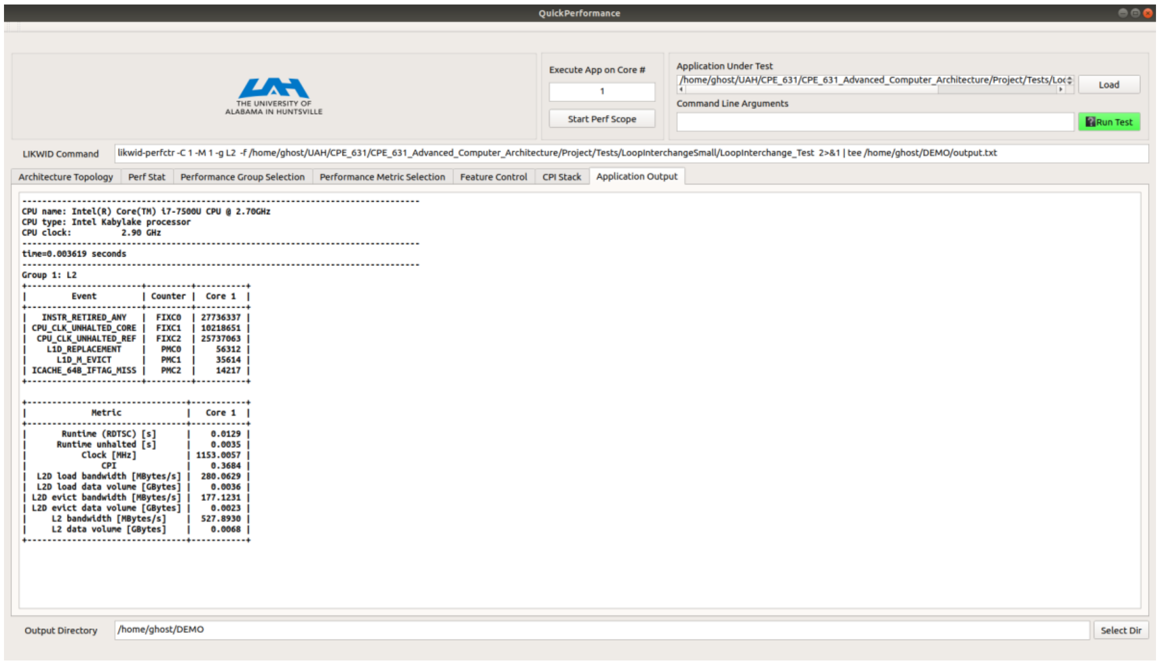
```
likwid-perfctr -C 1 -M 1 -g L2  -f /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/LoopInterchangeSmall/LoopInterchange_Base  2>&1 | tee /home/ghost/DEMO/output.txt
```

```
--------------------------------------------------------------------------------
CPU name:	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
CPU type:	Intel Kabylake processor
CPU clock:	2.90 GHz
--------------------------------------------------------------------------------
time=0.022956 seconds
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+-----------+
|         Event         | Counter |   Core 1  |
+-----------------------+---------+-----------+
|    INSTR_RETIRED_ANY  |  FIXC0  | 182182597 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 158981769 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 172587382 |
|    L1D_REPLACEMENT    |   PMC0  |  12175655 |
|      L1D_M_EVICT      |   PMC1  |   9996469 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |     14549 |
+-----------------------+---------+-----------+

+-------------------------------+-----------+
|             Metric            |   Core 1  |
+-------------------------------+-----------+
|      Runtime (RDTSC) [s]      |    0.0762 |
|      Runtime unhalted [s]     |    0.0547 |
|          Clock [MHz]          | 2674.9398 |
|              CPI              |    0.8727 |
|  L2D load bandwidth [MBytes/s] | 10230.3967 |
|  L2D load data volume [GBytes] |    0.7792 |
| L2D evict bandwidth [MBytes/s] | 8399.3710 |
| L2D evict data volume [GBytes] |    0.6398 |
|     L2 bandwidth [MBytes/s]   | 18641.9922 |
|     L2 data volume [GBytes]   |    1.4199 |
+-------------------------------+-----------+
```

Output Directory: /home/ghost/DEMO

Figure 3: Experiment L2 Cache Information



```
likwid-perfctr -C 1 -M 1 -g L2  -f /home/ghost/UAH/CPE_631/CPE_631_Advanced_Computer_Architecture/Project/Tests/LoopInterchangeSmall/LoopInterchange_Test  2>&1 | tee /home/ghost/DEMO/output.txt
```

```
--------------------------------------------------------------------------------
CPU name:	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
CPU type:	Intel Kabylake processor
CPU clock:	2.90 GHz
--------------------------------------------------------------------------------
time=0.003619 seconds
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+----------+
|         Event         | Counter |  Core 1  |
+-----------------------+---------+----------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 27736337 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 10218651 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 25737063 |
|    L1D_REPLACEMENT    |   PMC0  |    56312 |
|      L1D_M_EVICT      |   PMC1  |    35614 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |    14217 |
+-----------------------+---------+----------+

+--------------------------------+-----------+
|             Metric             |   Core 1  |
+--------------------------------+-----------+
|      Runtime (RDTSC) [s]       |    0.0129 |
|      Runtime unhalted [s]      |    0.0035 |
|          Clock [MHz]           | 1153.0057 |
|              CPI               |    0.3684 |
|  L2D load bandwidth [MBytes/s] |  280.0629 |
|  L2D load data volume [GBytes] |    0.0036 |
| L2D evict bandwidth [MBytes/s] |  177.1231 |
| L2D evict data volume [GBytes] |    0.0023 |
|     L2 bandwidth [MBytes/s]    |  527.8930 |
|     L2 data volume [GBytes]    |    0.0068 |
+--------------------------------+-----------+
```

Figure 4: Optimized Experiment L2 Cache Output

Using the L2 performance group we can see that the data volume flowing through L2 has gone down significantly as well with the optimized solution. Refer to the table below a tabularized view of the data gathered for this experiment.

Table 1: Loop Interchange 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. Overall, with the data gathered with the help of Quick Performance it would be safe to say that the new algorithm significantly reduces the use and traffic of the caches.

BLOCKING OPTIMIZATION

With blocking optimization, the developer will try to improve temporal locality to improve cache misses. The idea is instead of operating on entire rows or columns of an array, the blocking algorithm will work on submatrices or blocks. These submatrices will be of the size that they should fit in the cache and therefore all operations necessary should be carried out while the submatrix resides in the cache. In this experiment we explore matrix multiplication and the effect of varying block sizes. Refer to the appendix for the base and blocking snippets of code.

For the experiment we will be multiplying square matrices of size 1024x1024.

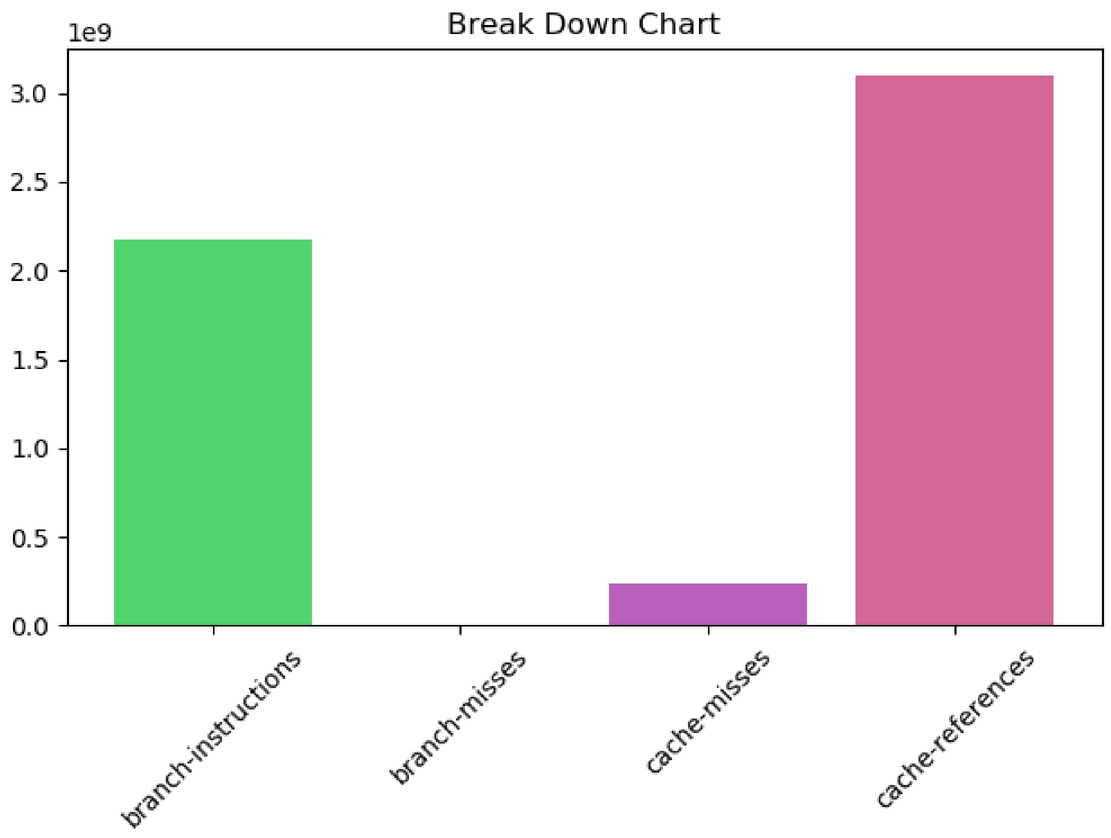


Figure 5: Matrix Multiplication Base Perf Output

Note that the branch misses are not zero, just low in consideration of the other metrics in the graph. This is why the perf stat tab allows the user to choose which metrics they would like plotted.

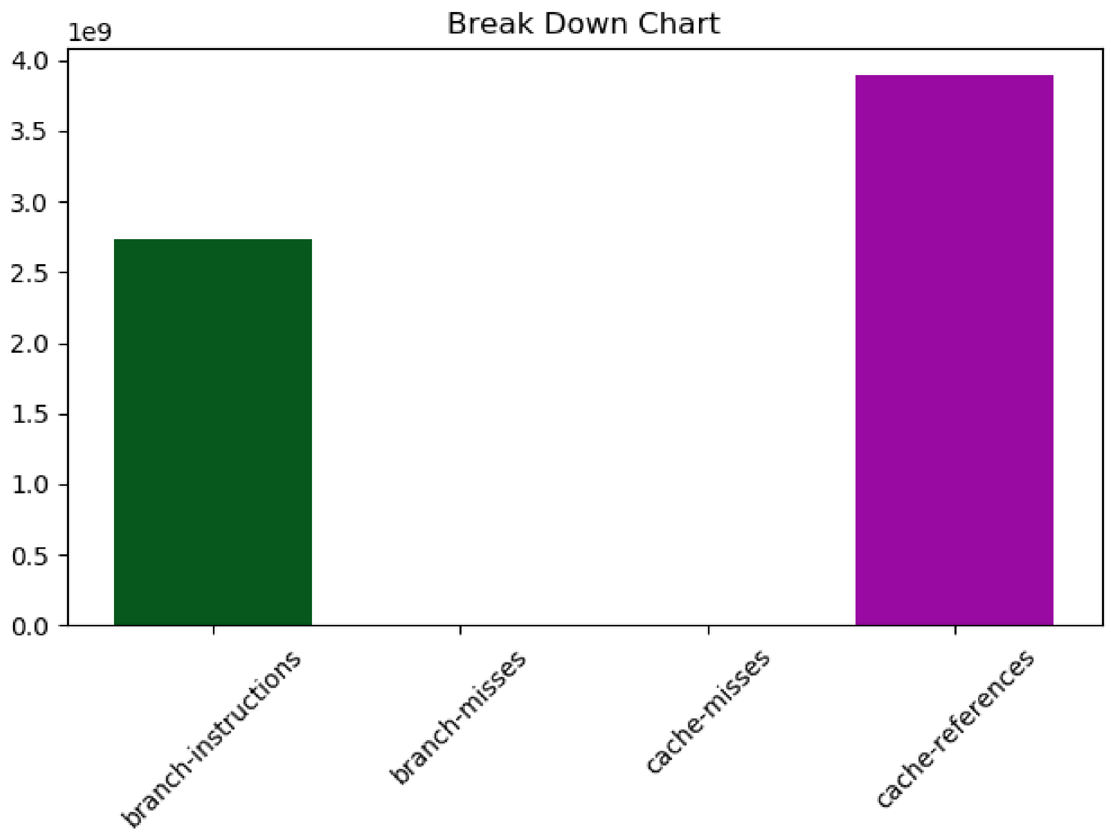


Figure 6: Matrix Multiplication Blocking (block size = 512) Perf Output

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
CPU type:       Intel Kabylake processor
CPU clock:      2.90 GHz
--------------------------------------------------------------------------------

time=11.325651 seconds
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+-------------+
|         Event         | Counter |    Core 0   |
+-----------------------+---------+-------------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 31348338539 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 35639871174 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 32812082072 |
|    L1D_REPLACEMENT    |   PMC0  |  1078803519 |
|      L1D_M_EVICT      |   PMC1  |     1686095 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |       70910 |
+-----------------------+---------+-------------+

+--------------------------------+-----------+
|             Metric             |   Core 0  |
+--------------------------------+-----------+
|       Runtime (RDTSC) [s]      |   11.3546 |
|      Runtime unhalted [s]      |   12.2724 |
|           Clock [MHz]          | 3154.3487 |
|               CPI              |    1.1369 |
|  L2D load bandwidth [MBytes/s] | 6080.6444 |
|  L2D load data volume [GBytes] |   69.0434 |
| L2D evict bandwidth [MBytes/s] |    9.5036 |
| L2D evict data volume [GBytes] |    0.1079 |
|     L2 bandwidth [MBytes/s]    | 6090.5477 |
|     L2 data volume [GBytes]    |   69.1559 |
+--------------------------------+-----------+
```

Figure 7: Matrix Multiplication Base L2 Cache LIKWID Output

This is the output generated from exercising the L2 performance group for the likwid-perfctr command.

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
CPU type:       Intel Kabylake processor
CPU clock:      2.90 GHz
--------------------------------------------------------------------------------
time=4.76542600 seconds
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+-------------+
|         Event         | Counter |    Core 0   |
+-----------------------+---------+-------------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 34634743461 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 16686671355 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 13896559963 |
|    L1D_REPLACEMENT    |   PMC0  |  1079191088 |
|      L1D_M_EVICT      |   PMC1  |     2340070 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |       23225 |
+-----------------------+---------+-------------+

+-------------------------------+------------+
|             Metric            |   Core 0   |
+-------------------------------+------------+
|      Runtime (RDTSC) [s]      |     4.7940 |
|      Runtime unhalted [s]     |     5.7458 |
|          Clock [MHz]          |  3487.2355 |
|              CPI              |     0.4818 |
|  L2D load bandwidth [MBytes/s] | 14407.2865 |
|  L2D load data volume [GBytes] |    69.0682 |
| L2D evict bandwidth [MBytes/s] |    31.2401 |
| L2D evict data volume [GBytes] |     0.1498 |
|     L2 bandwidth [MBytes/s]    | 14438.8367 |
|     L2 data volume [GBytes]    |    69.2195 |
+-------------------------------+------------+
```

Figure 8: Matrix Multiplication Blocking (block size = 512) L2 LIKWID Output

From the description of blocking optimization, it should cut down on the number of cache misses because the submatrix will be processed wholly while it resides in the cache. The figure below shows Cache Misses vs. Block Size used, note that block size 1024 has been removed because it incurred many misses which skewed the detail of the plot. Also, a block size of 0 in the plot represents the base case, this is just for formatting the plot and is **NOT** the blocking optimization program with zero block size.

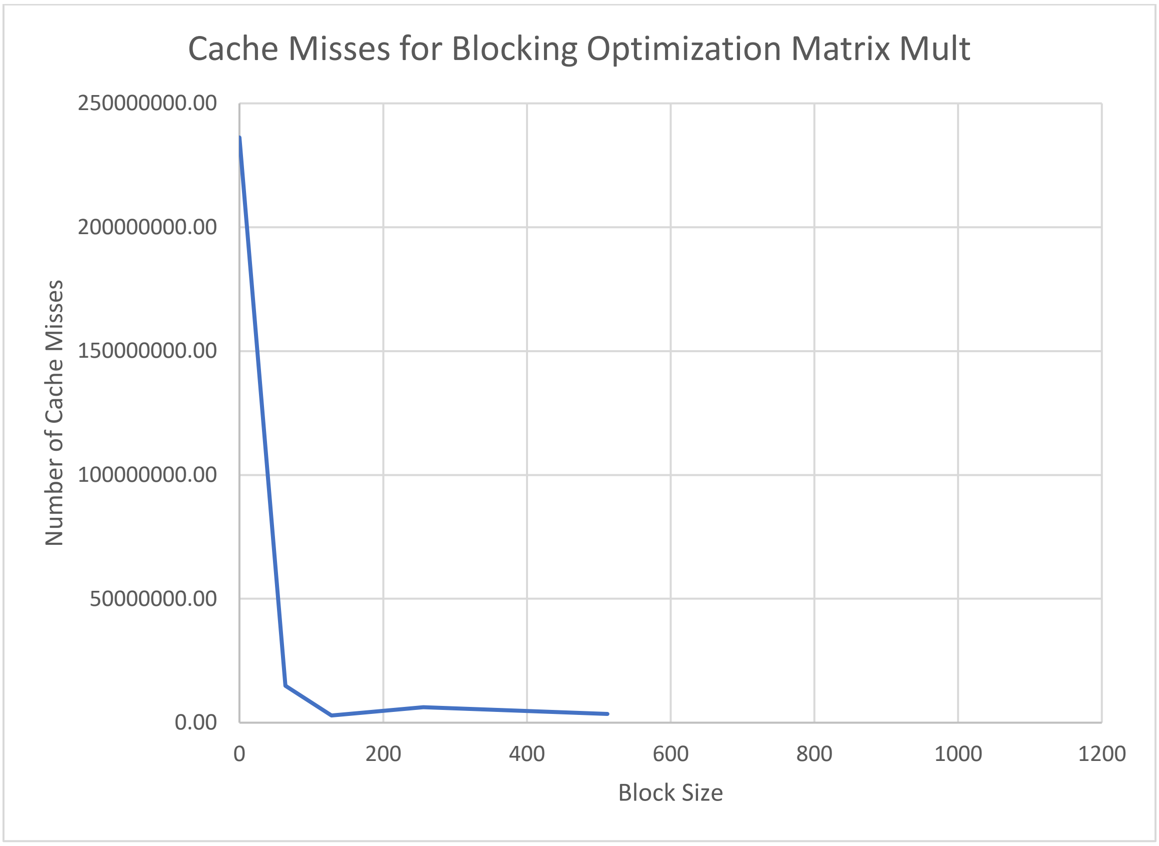


Figure 9: Cache Misses vs Block Size Blocking Optimization Matrix Mult

Block sizes 128 and 512 were close but using a block size of 512 resulted in the fewest cache misses for the multiplication. Because there were fewer cache misses the overall execution time of the application decreased significantly, refer to the figure below for Execution time in seconds vs Block Size.

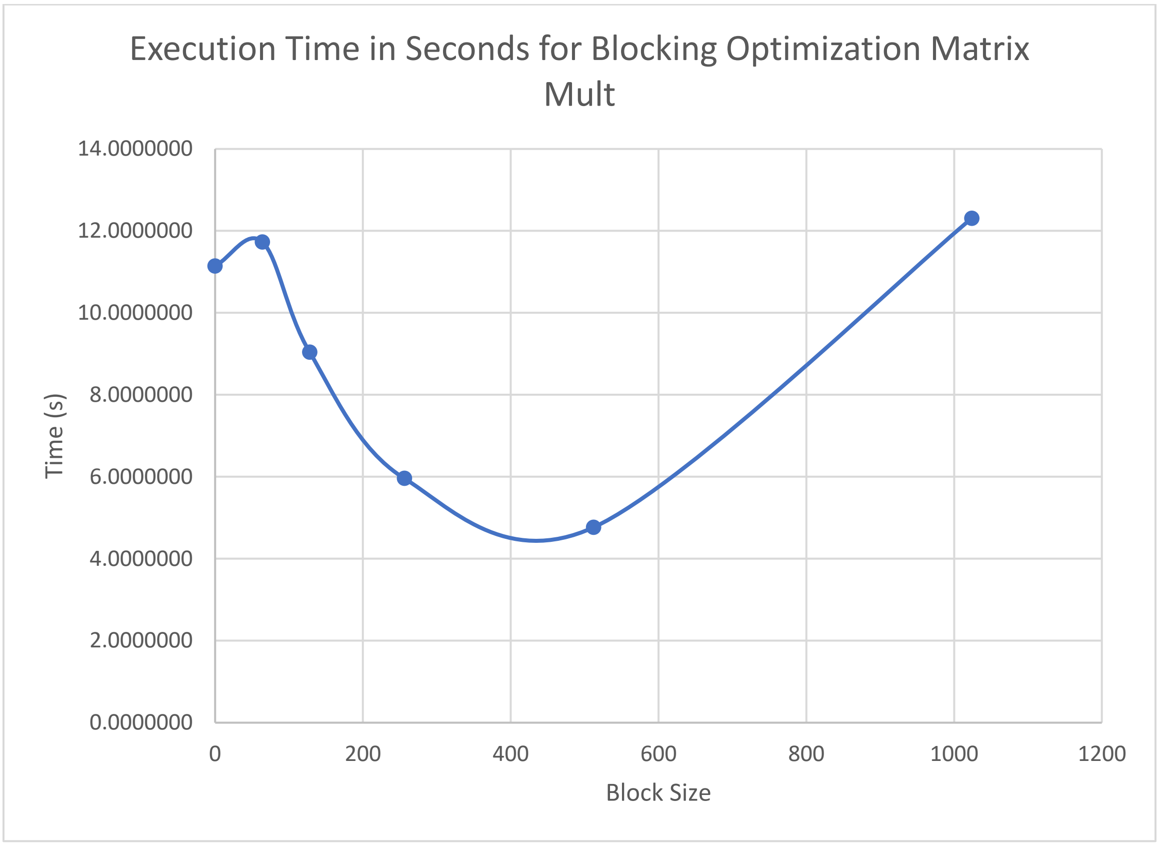


Figure 10: Execution Time vs. Block Size Blocking Optimization Matrix Mult

From the figure, it is clear that the block size of 512 provided the greatest improvement in execution time for this sample set of block sizes. Refer to the table below for all of the data gathered for this experiment.

Table 2: Blocking Optimization Matrix Multiplication Experimental Results

	Base	Block Size = 64	Block Size = 128	Block Size = 256	Block Size = 512	Block Size = 1024
Branch Instructions	2178244848.00	7034342057.00	4099742815.00	2730976847.00	2180261207.00	2177228234.00
Branch Misses	1176583.00	53845183.00	15881712.00	5373443.00	2186437.00	1187025.00
Cache References	3093737450.00	3662993733.00	5108623225.00	3889943321.00	3114454830.00	3104618073.00
Cache Misses	236232779.00	14903784.00	2904529.00	6249594.00	3493952.00	612175209.00
L2 Bandwidth (MB/s)	6090.55	19295.07	14492.66	14506.00	14438.84	5597.70
L2 Data Volume (GB)	69.16	226.82	131.44	86.92	69.22	69.05
L3 Bandwidth (MB/s)	15558.90	2464.60	29413.31	29472.28	29444.77	15844.37
L3 Data Volume (GB)	146.76	27.60	265.40	175.88	140.26	155.60
Time (seconds)	11.1462010	11.7281430	9.0394470	5.9622400	4.7654260	12.3077510

Conclusion

The purpose of this report was to introduce the developed tool, Quick Performance, and as a proof of concept demonstrate its usefulness in the software development cycle. This was tested by analyzing some simple software optimization techniques, loop interchange and blocking optimization, using the tool. Quick Performance allowed for quick loading and rapid performance metric feedback via the wrapped tools of Perf and LIKWID and the data gathered from the tool was used in the generation of tables and charts that further illustrate the optimization gained from these techniques. Quick Performance does not replace the more complex performance analysis tools, but it should alleviate some of the initial need for an easy to use performance analysis tool.

Future Work

Quick Performance has more potential to grow and there are many features of the underlying analysis tools, LIKWID and PERF, that were not incorporated into the final product. Below are a few additional areas where the tool could be improved.

1.) Modifying the tool to work with multiple cores at once
2.) Modifying the tool to work with multi-threaded applications
3.) Integrating LIKWID support for MPI applications
4.) Add a batching mode for users to queue up runs

Appendix

LOOP INTERCHANGE BASE

```
#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;
}
```

LOOP INTERCHANGE TEST

```
#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;
}
```

BASE MATRIX MULTIPLICATION

```
int main( int argc, char *argv[])
{
   float *a,*b,*c,dot_prod;
   int dim_l,dim_n,dim_m;
   int i,j,k;

   /*
   get matrix sizes
   */
   get_index_size(argc,argv,&dim_l,&dim_m,&dim_n);

   // dynamically allocate from heap the numbers in the memory space
   // for the a,b, and c matrices
   a = new (nothrow) float[dim_l*dim_m];
   b = new (nothrow) float[dim_m*dim_n];
   c = new (nothrow) float[dim_l*dim_n];
   if(a==0 || b==0 || c==0) {
     cout <<"ERROR:  Insufficient Memory" << endl;
     exit(1);
   }

   /*
      initialize numbers matrix with random data
   */
   srand48(SEED);
   fill_matrix(a,dim_l,dim_m);
   fill_matrix(b,dim_m,dim_n);


   /*
   Start recording the execution time
   */
   TIMER_CLEAR;
   TIMER_START;

   // multiply local part of matrix
   for (i=0;i<dim_l;i++) {
      for (j=0;j<dim_l;j++) {
         dot_prod = 0.0;
         for (k=0;k<dim_l;k++) {
            dot_prod += A(i,k)*B(k,j);
         }
         C(i,j) = dot_prod;
      }
   }

   /*
      stop recording the execution time
   */
   TIMER_STOP;

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

}
```

BLOCKING OPTIMIZATION MATRIX MULTIPLICATION

```
/*
Start recording the execution time
*/
TIMER_CLEAR;
TIMER_START;

float r = 0;
for (int jj = 0; jj < dim_l; jj = jj + block_size)
{
    for (int kk = 0; kk < dim_l; kk = kk + block_size)
    {
        for (int i = 0; i < dim_l; i = i + 1)
        {
              for (int j = jj; j < MIN(kk + block_size, dim_l); j = j + 1)
              {
                  r = 0;
                  for (int k = kk; k < MIN(kk + block_size, dim_l); k = k + 1)
                  {
                       r = A(i, k) * B(k, j);
                  }
                  C(i, j) = C(i, j) + r;
               }
         }
     }
}
```