CPEG 455 Lab 1

Shane Cincotta

10/21/19

Abstract:

The goals of this lab are split into 3 parts:

1. Gather a node profile, edge profile and path profile of a function.
2. Use the PAPI hardware counter library to measure the memory hierarchy performance (L1, L2 AND TLB cache misses) of the paths in the function.
3. Change the order of memory accesses to achieve a minimum and maximum L2 cache miss rate.

Detailed Strategy:

To begin, I had to gather three types of profiles of the function *func*.

<u>Node Profile Strategy</u>

Struct s1 has 5 attributes, *int a,b,c,d,e*. This is conveniently setup as a node so I will use an instance of this struct to represent my node (*s1 *node*). To complete my node, I needed to add 2 more attributes to *s1- int START* and *int END*. I then analyzed the control flow within *func* to create a total path.

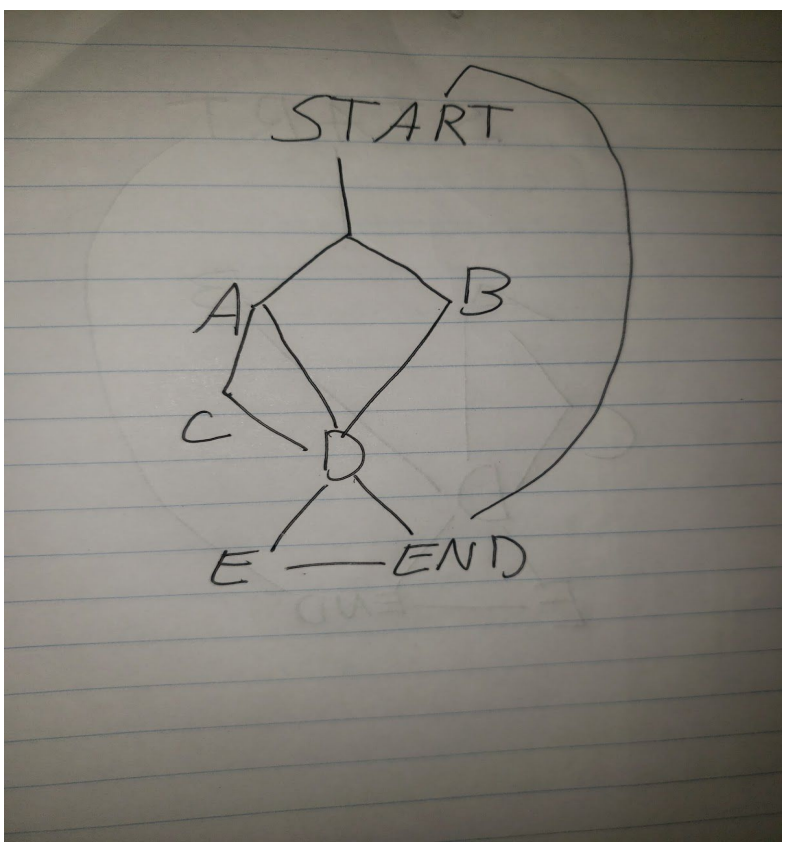


Fig 1. Total paths

Within *func,* nodes were inserted and incremented within each control flow statement whenever that branch was taken.

```
static inline void func(int a1, int a2, int a3)
{
    int i;
    node->START++;
    if(a1 || a3){
        //treat the for loops as nodes i.e we don't need to go into them to get the path
        node->a++;
        for(i=0; i<arraysize; i++){
            a[i].a += a[i].e;
        }
        if(a2){
            //A-C
            count+=2;
            node->c++;
            for(i=0; i<arraysize; i++){
                a[i].b++;
            }
        }
    }

    else {
        node->b++;
        //A-C
        count+=4;
        for(i=0; i<arraysize; i++){
            a[i].c = a[i].d++;
        }

    }
    node->d++;
    if(a2 && a3){
        //D-E
        count+=1;
        node->e++;
        for(i=0; i<arraysize; i++){
            a[i].b = a[i].d++;
        }
    }
    node->END++;
    //printf("%d", paths_list[count]);
    //printf("%s%d\n","Count: ", count);
```

Fig 2. Inserting and incrementing nodes

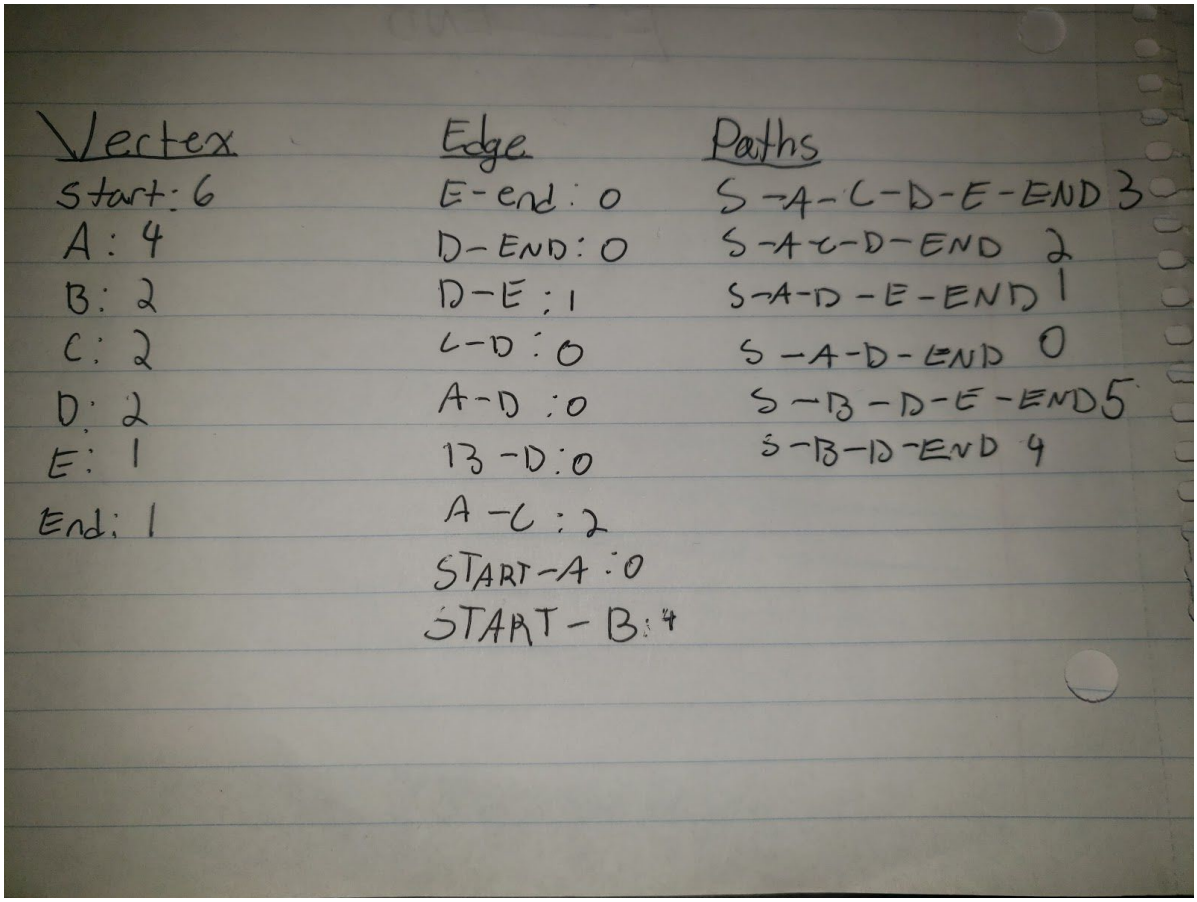
The values of the nodes were then printed out, representing how many times the nodes were accessed.

```
cincotta@cpeg655:~/CPEG455/Labs/Lab1$ gcc -I/usr/include prof.c -L/usr/lib -lpap
i
cincotta@cpeg655:~/CPEG455/Labs/Lab1$ ./a.out
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 1: NODE PROFILE
START: 10000
a: 7474
b: 2526
c: 3653
d: 10000
e: 2457
END: 10000
```

Fig 3. Results of running node profile on func w/ input 1

<u>Path Profile Strategy</u>

To gather the path profile, I first made a list of paths called *paths_list* which represents every possible path that can be taken. Then I used the path profiling algorithm as outlined in *lec5_profiling.pdf* to determine the sum of each individual path.



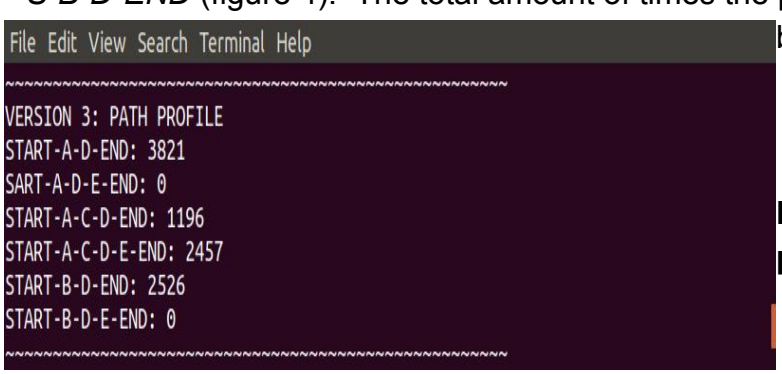
Vertex
Start: 6
A: 4
B: 2
C: 2
D: 2
E: 1
End: 1

Edge
E-end: 0
D-END: 0
D-E: 1
C-D: 0
A-D: 0
B-D: 0
A-C: 2
START-A: 0
START-B: 4

Paths
S-A-C-D-E-END 3
S-A-C-D-END 2
S-A-D-E-END 1
S-A-D-END 0
S-B-D-E-END 5
S-B-D-END 4

Fig 4. Using algorithm to calculate path sums

I then created a variable *int count* which represents the path sum. I used the results of the data in figure 4 to determine where and how much to increment *count* within *func* (figure 2 shows where *count* is incremented). The final value of *count* after each iteration of *func* is the index of *path_list* where the path resides. For example, if the value of *count* is 4, the path taken was *path_list[4]* which corresponds to path *S-B-D-END* (figure 4). The total amount of times the path was taken is then printed below.



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 3: PATH PROFILE
START-A-D-END: 3821
SART-A-D-E-END: 0
START-A-C-D-END: 1196
START-A-C-D-E-END: 2457
START-B-D-END: 2526
START-B-D-E-END: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fig 5. Results of running edge profile on func w/ input 1

Edge Profile Strategy

To begin, I determined every edge and created a list *edge_list* of the same size, each index represents a different edge. The results from the path profile were then used to construct the edge profile. The edge frequency was calculated by adding the path profile frequencies from every path which contains that specific edge. For example, to calculate the edge frequency for edge *START-A,* the profile frequencies of the paths which contain the edge *START-A* were added together. This number represents the total uses of edge *START-A.* The location within *edge_list* which contains that specific edge is then updated to contain the edge frequency for each edge.

```
File  Edit  View  Search  Terminal  Help
END: 10000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 2: EDGE PROFILE
S-A: 7474
S-B: 2526
A-C: 3653
C-D: 3653
A-D: 3821
B-D: 2526
D-E: 2457
D-END: 7543
E-END: 2457
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fig 6. Results of running path profile on func w/ input 1

Hardware counter strategy

The hardware counters were constructed by adding 3 events to *int events[]*, *PAPI_L1_DCM, PAPI_L2_DCM* and *PAPI_TLB_DM.* These 3 events correspond to level 1 data cache misses, level 2 data cache misses and TLB data cache misses. They were then added to the eventset in *main. PAPI_start and* PAPI_stop were then wrapped around the location where *func* is called. The results were stored in the list *values.*

```
File  Edit  View  Search  Terminal  Help
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
L2 CACHE MISS: 401
L1 CACHE MISS: 62425873
TLB CACHE MISS: 4751
cincotta@cpeg655:~/CPEG455/Labs/Lab1$
```

Fig 7. Result of cache misses on func w/ input 2

<u>Minimum and maximum L2 cache miss strategy</u>

To maximize the L2 cache miss rate, I implemented the stride *for loop* deoptimization technique utilized in the previous lab. I wrapped every *for loop* in *func* with a stride of 15.

```
File Edit Selection Find View Goto Tools Project Preferences Help
password.txt   prof.c   profUnoptimized.c
    node->START++;
    if(a1 || a3){
        //treat the for loops as nodes i.e we don't need to go into them to get the path
        node->a++;

        for(ii =0; ii <= stride ; ii++){
            for(i=0; i<arraysize; i+=stride){
                a[i].a += a[i].e;
            }
        }


        if(a2){
            //A-C
            count+=2;
            node->c++;

            for(ii =0; ii <= stride ; ii++){
                for(i=0; i<arraysize; i+=stride){
                    a[i].b++;
                }
            }

        }
    }

    else {
        node->b++;
        //A-C
        count+=4;

        for(ii =0; ii <= stride ; ii++){
            for(i=0; i<arraysize; i+=stride){
                a[i].c = a[i].d++;
            }
        }

    }
    node->d++;
    if(a2 && a3){
        //D-E
        count+=1;
        node->e++;

        for(ii =0; ii <= stride ; ii++){
            for(i=0; i<arraysize; i+=stride){
                a[i].b = a[i].d++;
            }
Line 222, Column 54                                                    Tab Size: 4   C
```

Fig 8. Using a stride to deoptimize for loops

This deoptimizes a *for loop* by reducing the effectiveness of spatial locality. Spatial locality refers to the repeated use of data elements within relatively close memory locations (cache). If the stride is long enough, the space between data elements becomes greater than that which is provided by spatial locality. The stride length necessary to completely deoptimize this code is 15. The deoptimization is now at a maximum, so I now need to optimize the L2 cache miss rate.

```
File  Edit  View  Search  Terminal  Help
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
L2 CACHE MISS: 5210
L1 CACHE MISS: 288916186
TLB CACHE MISS: 13248
cincotta@cpeg655:~/CPEG455/Labs/Lab1$
```

Fig 9. Result of cache misses w/ stride = 15

To minimize the L2 cache miss rate, I first need to eliminate my stride as well as parallelize my program. To parallelize my program I used *fork()*. I let the child process work on half the list, and the parent work on the other half.

```
File Edit Selection Find View Goto Tools Project Preferences Help
password.txt   sequential_access.c   profUnoptimized.c   prof.c
121     int i;
122     unsigned a1, a2, a3;
123
124     srandom(0);
125
126     //input1();
127     input2();
128     /* input2(); */
129     int pid = fork();
130     PAPI_start(eventset);
131     //child
132     if(pid ==0){
133         for(i=0; i<5000; i++){
134             a1 = (random() > t1);
135             a2 = (random() > t2);
136             a3 = (random() > t3);
137             func(a1, a2, a3);
138         }
139     }
140     //parent
141     else{
142         for(i=5000; i<10000; i++){
143             a1 = (random() > t1);
144             a2 = (random() > t2);
145             a3 = (random() > t3);
146             func(a1, a2, a3);
147         }
148     }
149     PAPI_stop(eventset, values);
150
151
Line 161, Column 50                                    Tab Size: 4    C
```

Fig 10. Implementing parallelization

```
File  Edit  View  Search  Terminal  Help
L2 CACHE MISS: 58
L1 CACHE MISS: 28875334
TLB CACHE MISS: 2271
cincotta@cpeg655:~/CPEG455/Labs/Lab1$
```

Fig 11. L2 cache miss with parallelization

Results:

<u>Profiling results</u>

Once all 3 profiling versions were constructed, they were run on *func* with 2 sets of input.

```
File  Edit  View  Search  Terminal  Help
cincotta@cpeg655:~/CPEG455/Labs/Lab1$ gcc -I/usr/include prof.c -L/usr/lib -lpap
i
cincotta@cpeg655:~/CPEG455/Labs/Lab1$ ./a.out
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 1: NODE PROFILE
START: 10000
a: 7474
b: 2526
c: 3653
d: 10000
e: 2457
END: 10000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 2: EDGE PROFILE
S-A: 7474
S-B: 2526
A-C: 3653
C-D: 3653
A-D: 3821
B-D: 2526
D-E: 2457
D-END: 7543
E-END: 2457
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 3: PATH PROFILE
START-A-D-END: 3821
SART-A-D-E-END: 0
START-A-C-D-END: 1196
START-A-C-D-E-END: 2457
START-B-D-END: 2526
START-B-D-E-END: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
File  Edit  View  Search  Terminal  Help
VERSION 1: NODE PROFILE
START: 10000
a: 8454
b: 1546
c: 3349
d: 10000
e: 836
END: 10000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 2: EDGE PROFILE
S-A: 8454
S-B: 1546
A-C: 3349
C-D: 3349
A-D: 5105
B-D: 1546
D-E: 836
D-END: 9164
E-END: 836
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
VERSION 3: PATH PROFILE
START-A-D-END: 5105
SART-A-D-E-END: 0
START-A-C-D-END: 2513
START-A-C-D-E-END: 836
START-B-D-END: 1546
START-B-D-E-END: 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fig 12. Running 3 profiles on 2 data sets

Hardware counter results

As shown in figure 7, the initial L2 cache miss rate was ~400. The stride deoptimization technique did in fact increase the L2 cache miss rate, now I had to deoptimize it enough to cause the maximum amount of cache misses. To do this, I experimented with different stride values. I found that a stride of 15 caused the most amount of cache misses (~5k), this can be compared to the cache misses without a stride (~400). I found that if I increased the stride above 16, the L2 cache miss rate would reduce. This is most likely due to spatial locality. The processor assumes that programs will take use of spatial locality, and thus if an element is accessed, there is a high probability that the following data elements will be accessed. To prepare for this, the CPU loads the next 15 data elements into the cache whenever a specific element is accessed. Thus by having a stride of 15 the processor will load the first element accessed, as well as the succeeding 15 elements, but the stride of 15 skips the 15 succeeding elements and thus they never get accessed. Thus all optimization due to spatial locality is eliminated.

The results of deoptimizing the L2 cache miss rate was successful. Using parallelization, the L2 cache miss rate dropped to ~60 (fig 11). This is because the cache hierarchy is capable of carrying out multiple memory requests. By using parallelism, a virtual cache is created giving the illusion to the processes that they each have their own dedicated cache. These concepts combined with the reduced *for loop* length (half the length for each process) result in a lower L2 cache miss rate. To increase the L2 cache miss rate further, one could use multiple instances of *fork()*.

Conclusion:

The results of this lab were largely successful. A node profile, edge profile and path profile were derived from the function *func.* It depicts the frequency of each node, edge as well as path.

Hardware counters were also successfully implemented to record the number of L1, L2 and TLB cache misses. The function *func* was then optimized and deoptimized to result in the minimum and maximum L2 cache misses. It was deoptimized using a stride of length 15. It was optimized by implementing parallelism. Both of these techniques were successful and increasing and decreasing the number of L2 cache miss.