CPEG655

Lab 02

Michael Hutts

System Description

Work on this project was done on two systems, the details are as follows:

System 1:

Operating System: Ubuntu 22.04.2 LTS (Using WSL under Microsoft Windows 10 Enterprise 10.0.19045 N/A Build 19045)

CPU Info: Intel Core i7-8550U

CPU Details: See *cpu_details_system_1.txt*

Compiler: gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

Problem 1: Transform Code Guided By Hardware Counters

1.1 Redesign the struct "Mem"

On my system, cache line size is 64 bytes. A char uses 1 byte, and an int uses 4 bytes.

Original Structure of Mem:

int fa;

char fb;

int fc;

char fd;

int fe;

fa	fb	fc	fd	fe
4 bytes	1 byte	4 bytes	1 byte	4 bytes

Total Size: 14

Mem fills up a cache of size 64, 4 times over in full, with 8 additional bytes remaining

In this case, part of the 5th Mem will exist on one line while the other part on a second line. This is an issue with the current structure as for *a[i].fa = a[i].fb+a[i].fd;*, the data will be split between these two cache lines as in the first cycle fa and and fd are on different cache lines. It's also cut off at an odd size compare to what the cache line can fit, so it wastes cacheline space compared to squeezing the necessary data for the loops into the same cache line consistently.

Modified Structure of Mem:

The way to optimize this solution is to join these values together in a cache-optimal manor, specifically like so:

int fc;

int fe;

int fa;

char fb;

char fd;

Putting fc and fe at the front showed marginal improvement (about 9700000 vs 11700000). Moving these values to the end showed a very similar performance change, which makes sense to me since the order simply determines if the first or the second cache line will be completely filled (one line will have 56+8 bytes while the other will have 56+6 bytes).

New pattern for filling cache:

14 bytes	14 bytes	14 bytes	14 bytes	4 bytes	4 bytes	
Mem	Mem	Mem	Mem	fc	fe	
4 bytes	1 byte	1 byte	14 bytes	14 bytes	14 bytes	14 bytes
fa	fb	fd	Mem	Mem	Mem	Mem

A solution that I believe would be even better would be to split fc and fe, and fa, fb, fd into their own arrays of length len. I think this would be permissible within the restriction and show a far more dramatic improvement, but I really struggled with getting the syntax of this correct, so my end solution may be suboptimal, but it does fully fit the rules and work completely. In 1.2 I applied a more extreme version of this concept, where the values are completely split up as variables to take advantage of the greater freedom offered in 1.2.

1.2 Co-optimize the struct "Mem" and the function "func"

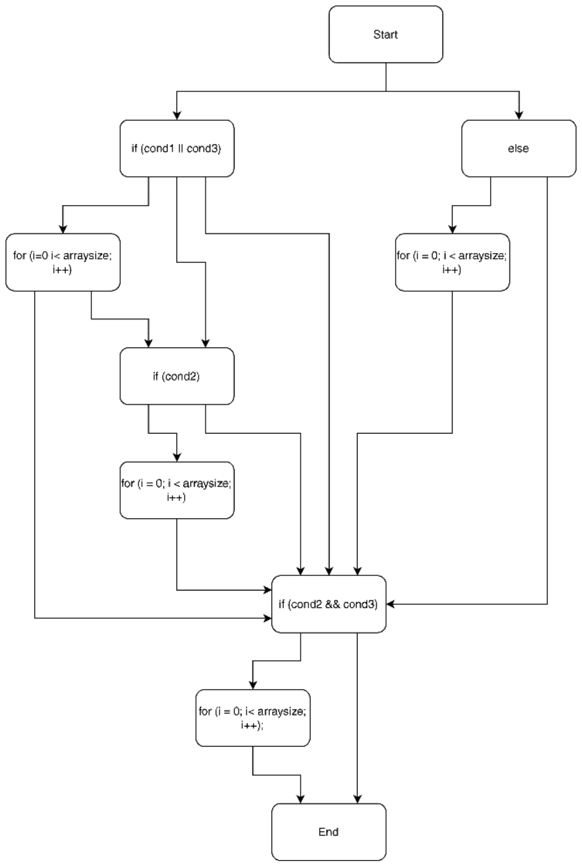
In this version I've transformed Mem into a similar structure to a Struct of arrays. Unfortunately, I was having issues with getting the syntax correct to do this as a single Mem struct, (see my last note on 1.1), so I've split it into two dedicated structs, which are then individually split into arrays of *len* elements.

This is probably the most optimal way of performing this while maintaining the same overall load. The cache is saturated with only the relevant values for the entire duration of each loop. This gave a dramatic decrease in cache misses (about 100x less), at an average of about 850000 vs the original 11700000.

Problem 2: Node Profile, Edge Profile, and Path Profile

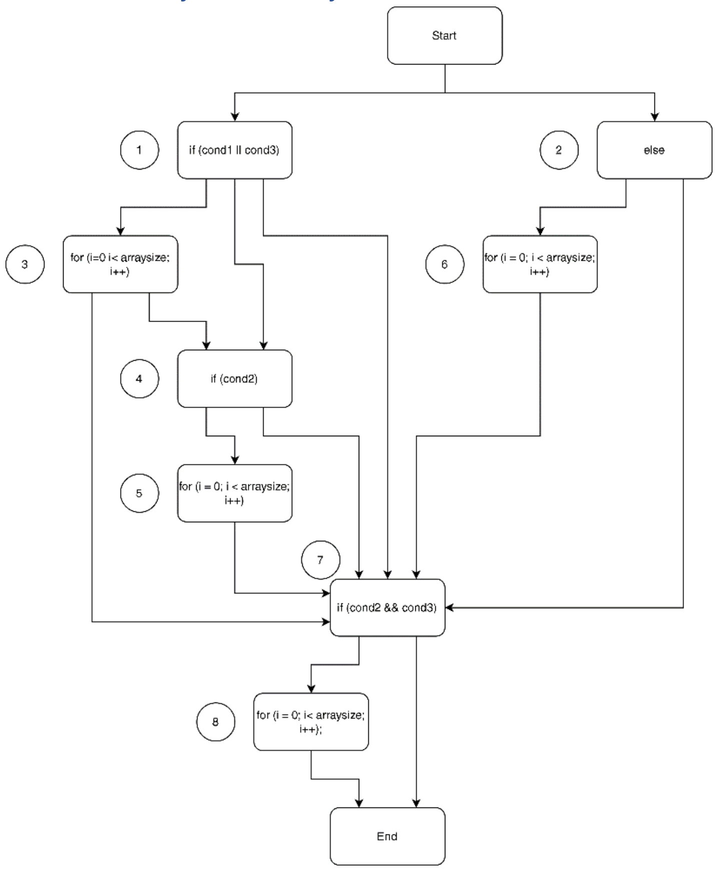
2.1 Profiling

Control Flow Graph:


Start
if (cond1 || cond3)
else
for (i=0 i< arraysize; i++)
for (i = 0; i < arraysize; i++)
if (cond2)
for (i = 0; i < arraysize; i++)
if (cond2 && cond3)
for (i = 0; i< arraysize; i++);
End

Node Profile:

Labeled Nodes for Node Profile:


Start
1
if (cond1 || cond3)
2
else
3
for (i=0 i< arraysize: i++)
6
for (i = 0; i < arraysize; i++)
4
if (cond2)
5
for (i = 0; i < arraysize; i++)
7
if (cond2 && cond3)
8
for (i = 0; i< arraysize; i++);
End

Results:

Input 1:

Node Traversal:

Node 0: 7474

Node 1: 2526

Node 2: 122454016

Node 3: 3653

Node 4: 59850752

Node 5: 41385984

Node 6: 2457

Node 7: 40255488

Input 2:

Node Traversal:

Node 0: 8454

Node 1: 1546

Node 2: 138510336

Node 3: 3349

Node 4: 54870016

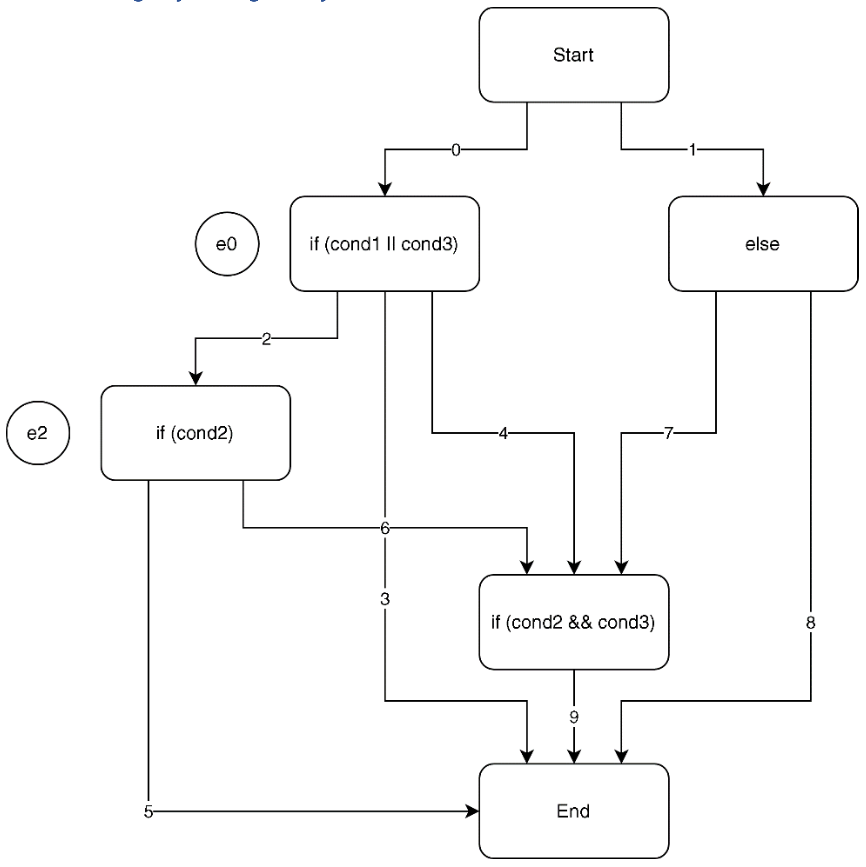
Node 5: 25329664

Node 6: 836

Node 7: 13697024

Edge Profile:

Labeled Edges for Edge Profile:



Input 1:

Edge Traversal:

```
Edge 0: 7474

Edge 1: 2526

Edge 2: 3653

Edge 3: 6347

Edge 4: 0

Edge 5: 1196

Edge 6: 2457

Edge 7: 0

Edge 8: 0

Edge 9: 2457
```

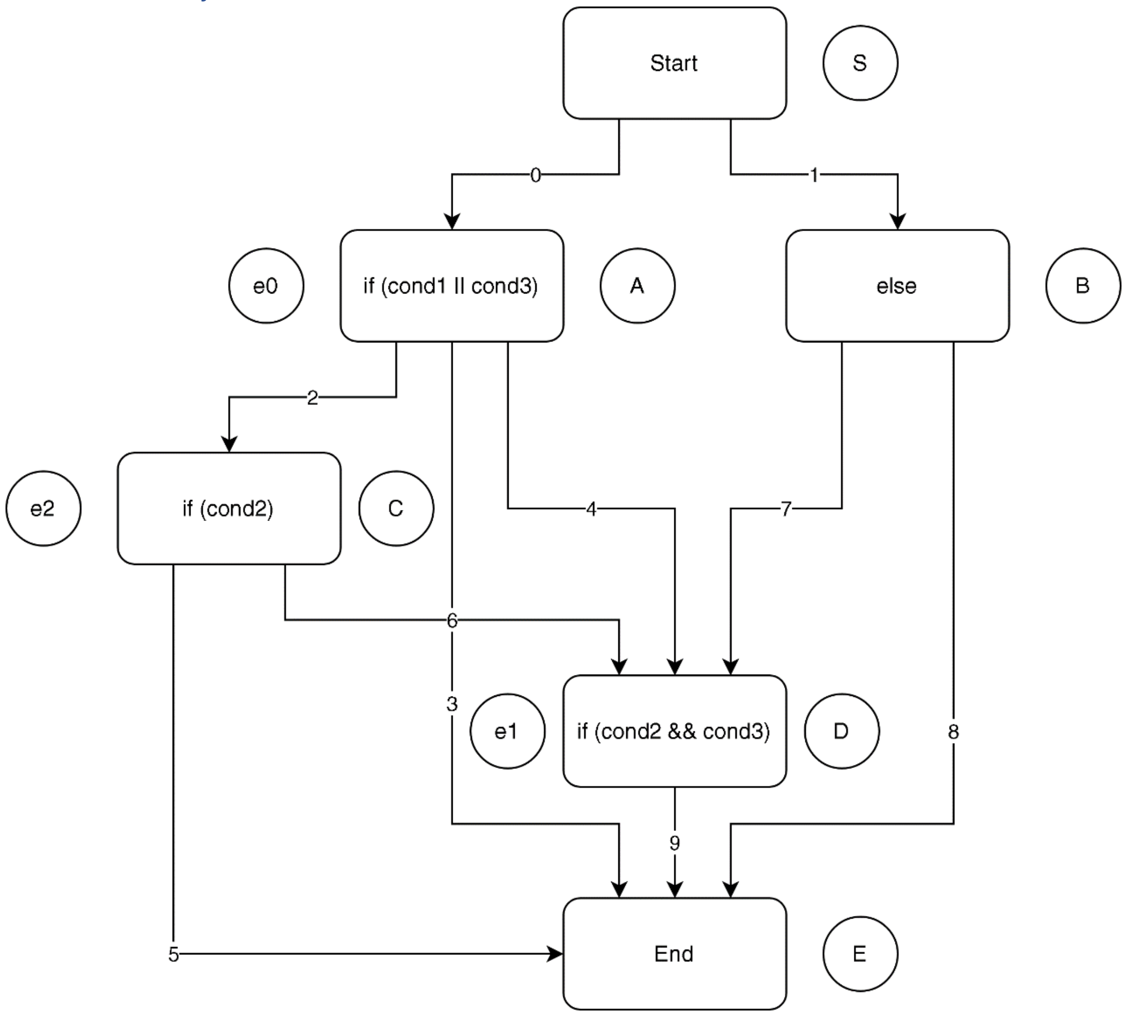
Input 2:

Edge Traversal:

```
Edge 0: 8454

Edge 1: 1546

Edge 2: 3349

Edge 3: 6651

Edge 4: 0

Edge 5: 2513

Edge 6: 836

Edge 7: 0

Edge 8: 0

Edge 9: 836
```

Path Profile:

Labeled Path Profile:



All possible paths:

paths[0] = SACDE (e0, e2, e1)

paths[1] = SACE (e0, e2)

paths[2] = SADE (e0, e1)

paths[3] = SAE (e0)

paths[4] = SBDE (e1)

paths [5] = SBE

Input 1:

Path Traversal:

Path 0: 3653

Path 1: 0

Path 2: 6347

Path 3: 0

Path 4: 0

Path 5: 0

Input 2:

Path Traversal:

Path 0: 3349

Path 1: 0

Path 2: 6651

Path 3: 0

Path 4: 0

2.2 Practice Using Hardware Counters and Optimize for Memory Hierarchy

(1)

Input 1 Average over 250 Runs: 126177588.544

Input 2 Average over 250 Runs: 108209129.188

(2)

We saw in in our path profiling that our code actually only takes two paths. One where cond3 is true, and one where it isn't. The path of *if(cond1 || cond3)* is always executed, as is path of *if(cond2)*, therefore the only real conditional in our code is whether or not *if(cond2 && cond3)* is true. I initially observed that in this case the for loop that performs "a[i].c++" is actually redundant, but we can't remove memory accesses in the actually executed sequences. Therefore, this redundant operation must remain. We can however neglect the paths that are never taken since we only are concerned with the two sets of inputs provided. With this in mind, a significant amount of the logic in func() can be gutted, and s1 can be rearranged to accommodate the access order of s1 (for(a += e; c++;), for(c=d++;)). In other words, the relevant pairs of values can be grouped together in s1 and unexecuted paths in func() can removed to minimize L2 cache misses.

Input 1 Average Over 250 Runs: 96034405.396

Input 2 Average Over 250 Runs: 83573315.368