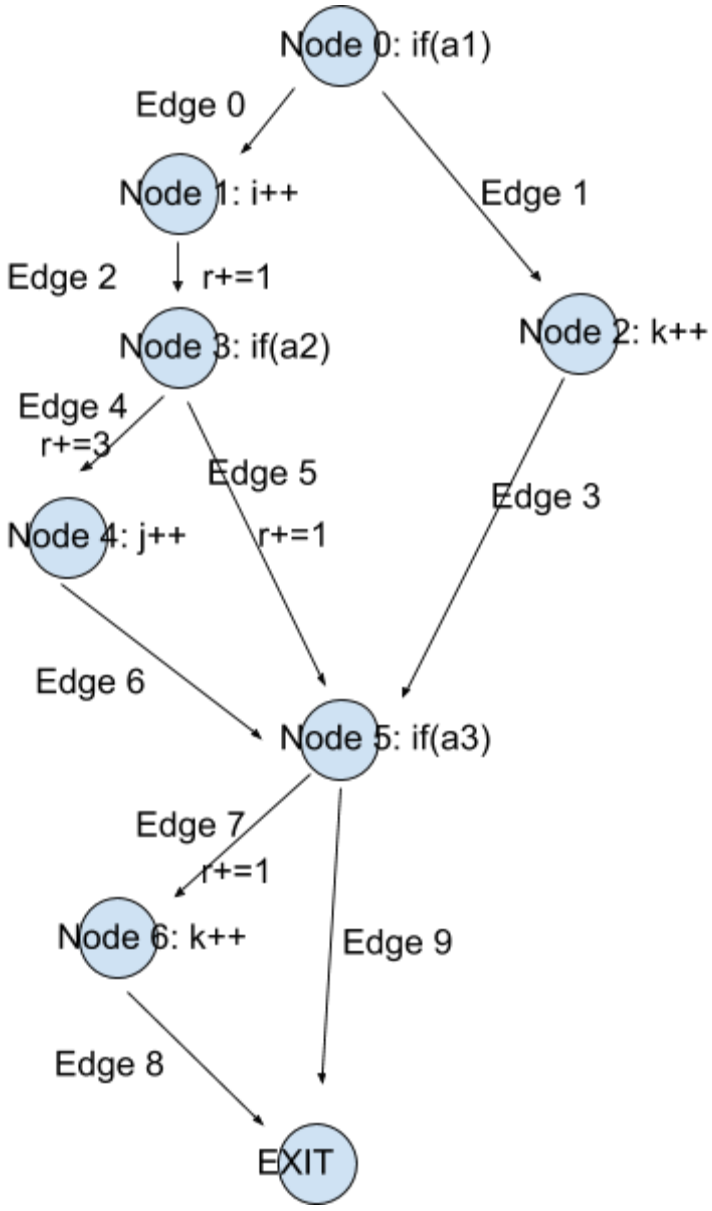
Katherine Perry
CPEG 455
23 October 2020

Lab #2: Program Profiling and Profile Enabled Optimization

Corresponding C Files:
Node Profile: prof_node
Edge Profile: prof_e
Path Profile: prof_p

	Run 1	Run 2
Node	Freq[Node]	Freq[Node]
0	10000	10000
1	4871	4933
2	5129	5067
3	4871	4933
4	2403	2522
5	10000	10000
6	4977	4917

	Run 1	Run 2
Edge	Freq_e[Edge]	Freq_e[Edge
0	5029	4980
1	4971	5020
2	5029	4980
3	4971	5020
4	2525	2533
5	2504	2447
6	2525	2533
7	5106	4990



8	5106	4990
9	4894	5010

		Run 1	Run 2
Path	code	Freq_p[r]	Freq_p[r]
0-2-5-EXIT	r=0	2549	2474
0-2-5-6-EXIT	r=1	2487	2520
0-1-3-5-EXIT	r=2	1263	1255
0-1-3-5-6-EXIT	r=3	1221	1248
0-1-3-4-5-EXIT	r=4	1231	1248
0-1-3-4-5-6-EXIT	r=5	1249	1255

*Edges that were weighted :edge 2 =+1, edge 4=+3, edge 5=+1 edge 7= +1

Optimizing Data Layout and Code Structure

3.1 Corresponding C files: mem_l1.c and mem_tlb.c

	Original code	Combined into one for loop	Second for loop listed before the first Loop	Loop one nested inside of Loop two	Loop two nested inside of Loop one
L1 miss	16213516 (16052383)	5767964	17311890 (16335133)	15302412	15660316
TLB miss	786604	464050	817773	578480	563850

To maximize the amount of L1 cache misses and TLB misses I edited the function “func” many times and the versions that maximized it the most are listed above in the table. In order to increase cache misses, cache conflicts would have to be created. I created cache conflicts by changing the order of the for loops and by changing the location of the increment variable “i”. After running several tests, the data showed having the second for loop listed first as well as having the increment variable initialized in the for loops, increase cache misses and TLB misses the most. This is because we are loading the most cache lines in this version of “func” than any

other version as well accessing “i” to cause interruptions which will lead to more cache misses. The values in parentheses show the misses before changing the initialization of “i’. This code also accesses the struct variables in such a way that would increase cache miss, in an order that neglects spatial locality.

3.2 Corresponding C files: mem_struct_p1.c (ran V1 and V2) and mem_struct_p2.c (ran V3)

	V1: orignal	V2: order change	V3: separate structs
Cycles	2502799971	2346949432	1912986750
L1 miss	16240017	15638614	7452250

To minimize the code’s L1 cache misses I created three different versions of the struct initializations for the function “func”. The second and third versions are the versions of code with adaptations. The second version changes the listing of the variables to match the order “func” loads and stores them from the stack. The third version does the same thing as well as creating two different structs to designate the variable to their corresponding cache line that will be accessed by “func”. Both version two and three decrease the amount of L1 cache misses by changing the instance at which they are accessed. Version three minimizes L1 cache misses the most. This is because the order of the variables matches the order “func” accesses them, and the struct declaration allows a continual call to a specific cache line until the code is ready to move onto the next. These decrease the amount of L1 cache misses because the machine is accessing the needed cache line more frequently.

Note: v3 was gathered using mem_struct_p2 but is shown simplified in mem_struct_p1(just wouldn't run)

3.3 Corresponding C files: mem3_3.c

	cycles	L1 miss
Original version	18647590200	73808720
Optimized code	10544163738	26966395

Using the data from table one and table two, I created a new version of code that optimized prof1.c by changing the struct declaration as well as “func”. In the first table, the data showed that having the second Loop first would increase overall misses. It also showed that nesting the first loop inside the second loop minimizes cache misses. By using this method and the same struct declaration as I did in version three in the table at the top of the page, the data above shows a decrease in L1 cache misses. This is because the order of the struct declaration works with “func” to designate the variable to the corresponding cache lines for each loop in “func”. By optimizing the loops and struct initializations, the L1 cache misses minimize.