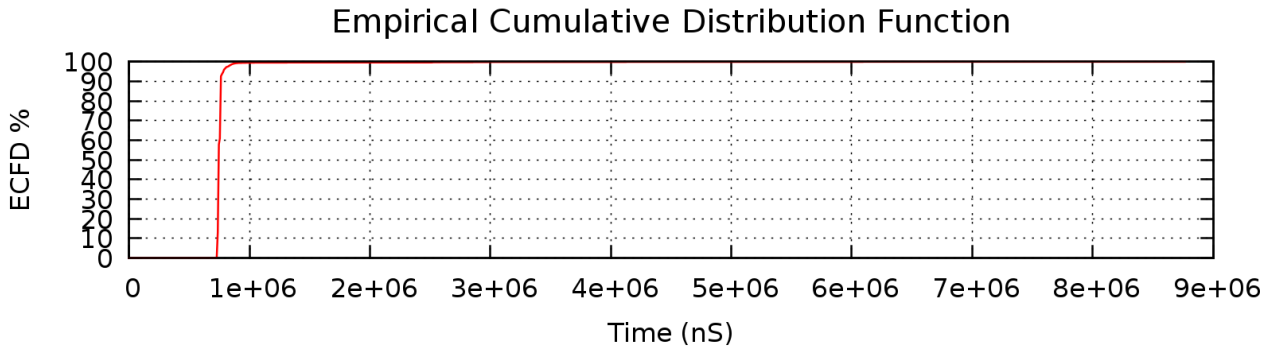
Eric Hair
Shakti Chaganty
Brent Mellor

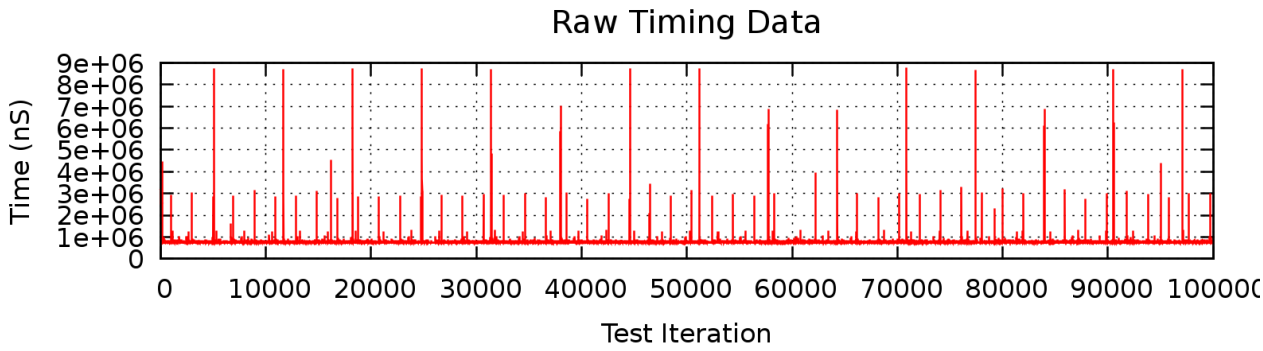
ECE 6785 – Advanced Embedded Systems
Team Assignment 3

Timing Analysis

In order to test our DECIDE function we wrote a simple test harness which monitored the duration of the function call using the clock_gettime function. After collecting the raw timing data of 100,000 iterations, the test harness calculated the empirical cumulative distribution function. Both sets of data were plotted together using gnuplot.

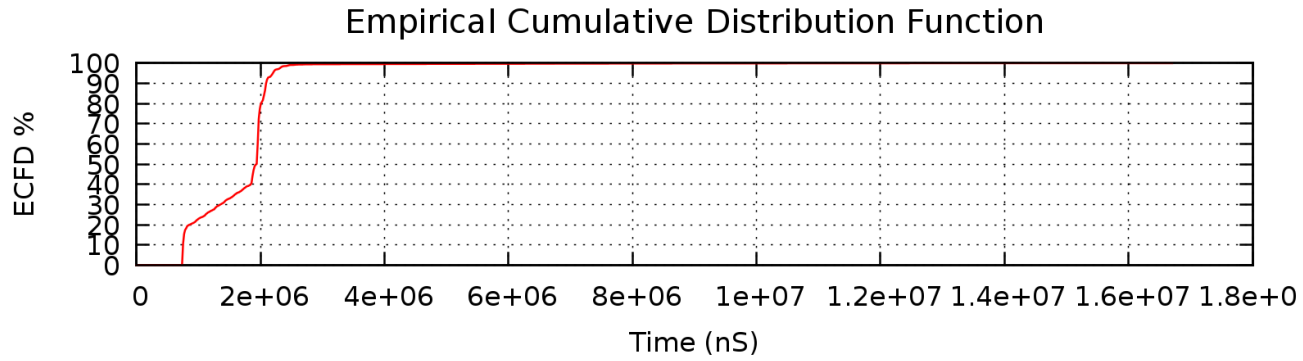
Unloaded Operation: Test duration ≈ 75 seconds

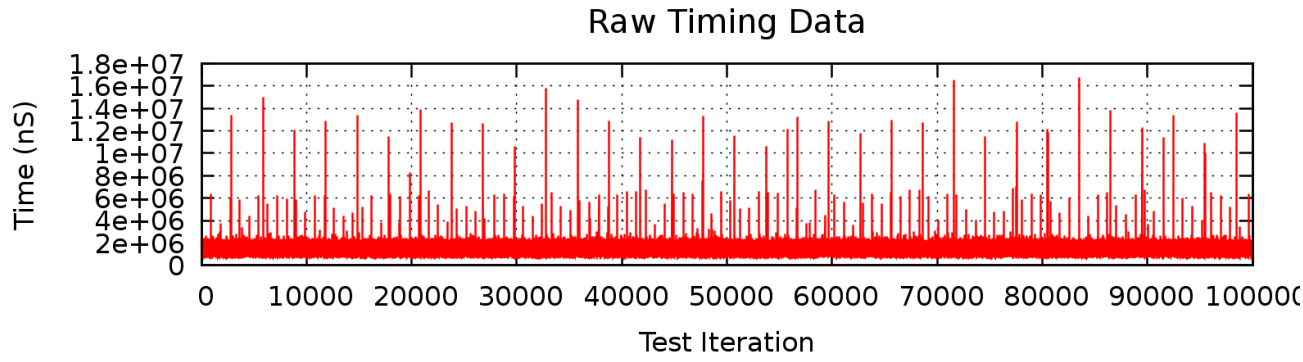




Loaded Operation: Test duration ≈ 168 seconds

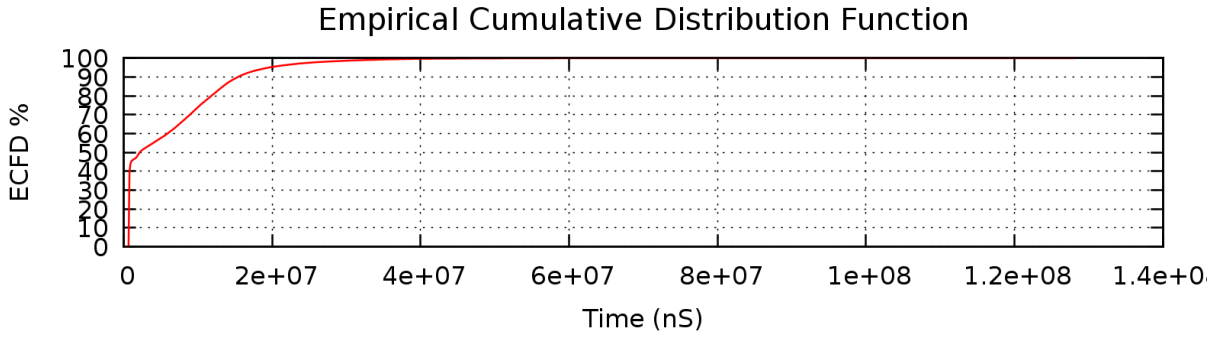
System was loaded by iperf over the 10/100MB eth0 interface.

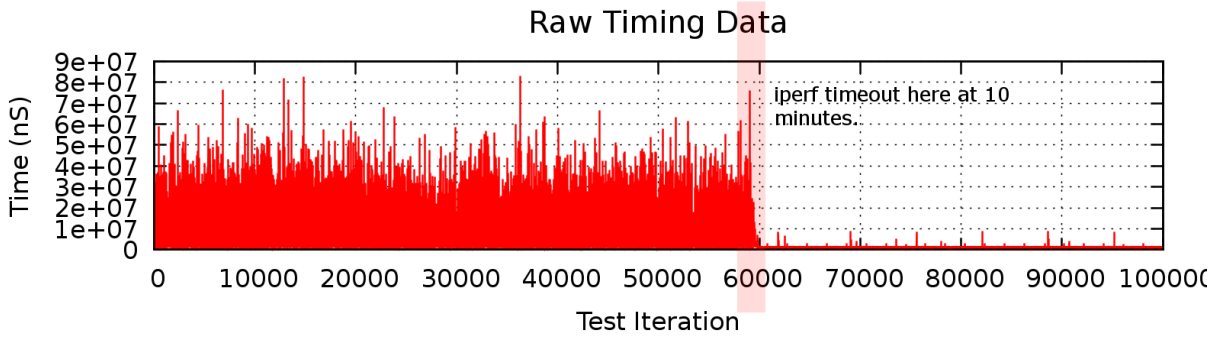




Mega Loaded Operation: Test duration > 10 minutes

System was loaded by iperf over the eth0, usb0, and wlan0 interfaces. (For purely entertainment purposes…)





Timing Discussion

As seen in the raw timing data, the loaded BeagleBone requires significantly larger amounts of time in order to complete each iteration of the DECIDE function. This has the effect on changing the knee of the resulting CDF to have a shallower curve and mid–rise plateau.

The CDF plots show the average distribution of the times required to iterate. The unloaded plot shows a very rapid rise indicating that the execution times were very either low or high extremes with few mid-range values. By looking at the position of the CDF curve and raw timing data, we can assume that the average execution time was very small with regular peaks caused by occasional system interrupts.

In contrast, the loaded CDF shows that many iterations were interrupted by rapid-fire interrupts or thread context switches due to the large amounts of Ethernet traffic generated by iperf. Looking at the plateau in the curve, we can assume that while some iterations completed uninterrupted and others were delayed by long system tasks, the vast majority were delayed to a mid-range value by the Ethernet system.

Timing Improvements

Because Linux is a multi-threaded context switching environment, it has a complex system for determining the amount of time allocated for each process. Depending on the relative importance of the code being run within a process, one possible way to increase performance under heavy load is to elevate the processes importance level in the system timeslot scheduler. This way, more time is allocated directly while less-important tasks suffer.

A second way to increase performance would be to enlarge the I/O buffer on the Ethernet system. Although there is still the necessity for rapid interrupts to collect data from the physical device, the amount of work done per interrupt can be lessened as more data processing can be moved to a separate task. The larger the I/O buffer is for a device, the less often and perhaps more efficiently the processing task can be run.

Finally, one of the most difficult but potentially more beneficial improvements is to increase the memory and cache efficiency of your program. While you may not be able to prevent the system from switching away from your process, you can improve the cost involved in loading and storing your program’s context. This also has advantages during non-loaded execution time.

Stack Usage Analysis

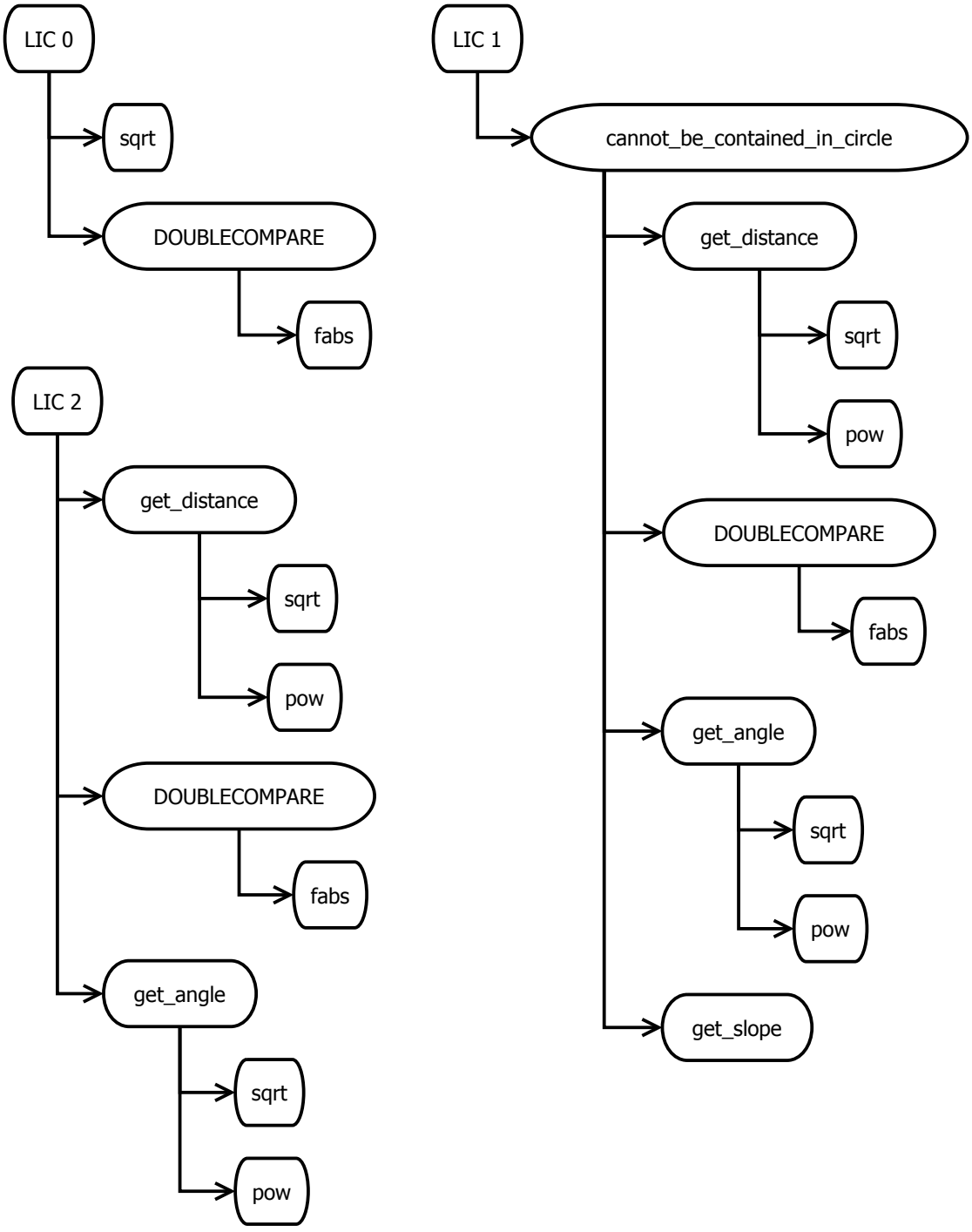
By disassembling our compiled code we were able to determine the stack usage of the functions used by each of our launch intercept conditions and support functions. By walking through the C-code, we mapped out each LIC function as it called additional functions and system libraries. Included below is a table summarizing the data we collected and showing the stack usage of each of the functions:

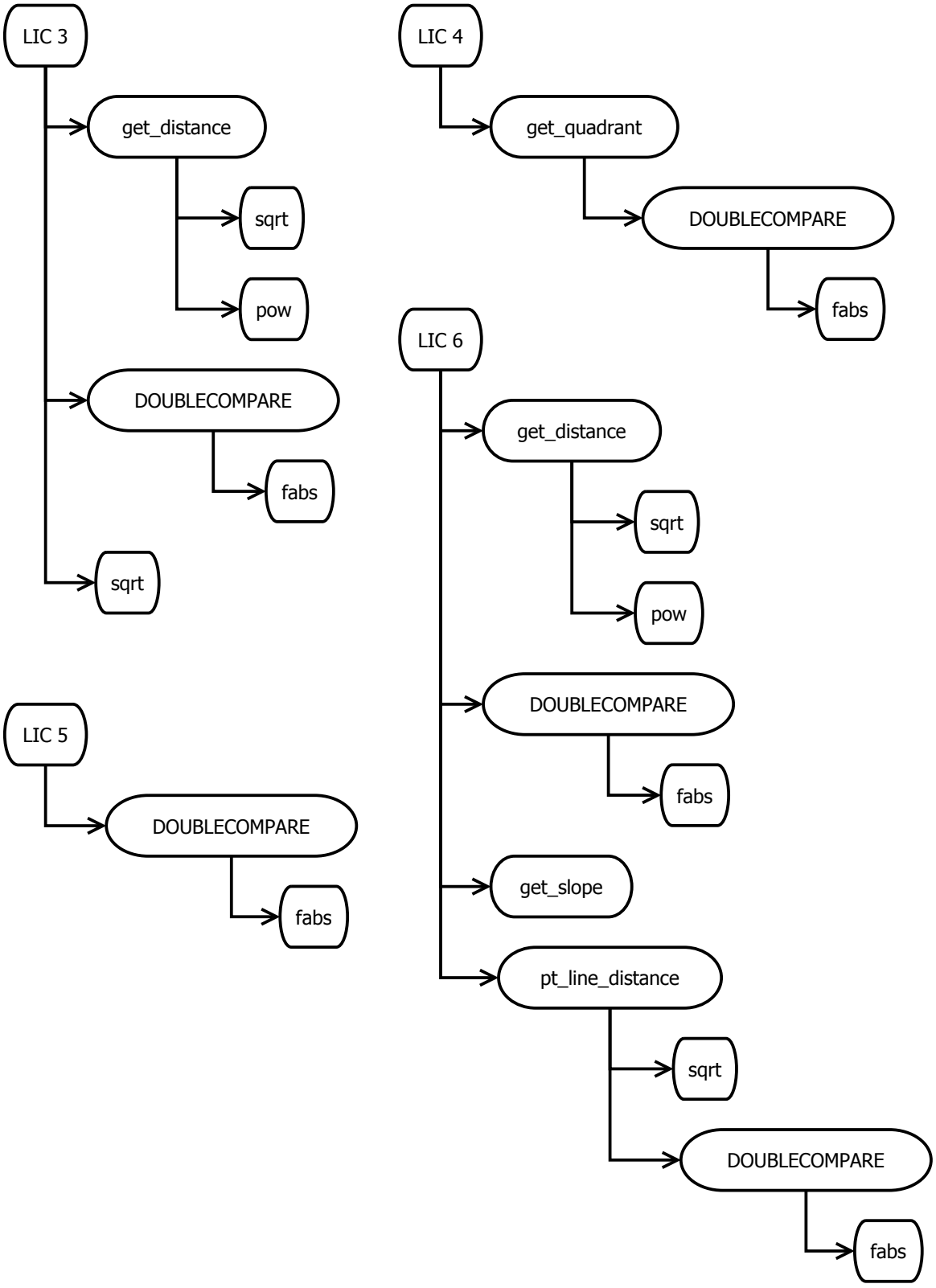
Function name	Number of bytes used on stack
DECIDE	40
LIC_0	156
LIC_1	96
LIC_2	416
LIC_3	352
LIC_4	132
LIC_5	104
LIC_6	412
LIC_7	164
LIC_8	104
LIC_9	416
LIC_10	344
LIC_11	112
LIC_12	284
LIC_13	104
LIC_14	648
cannot_be_contained_in_circle	816
get_quadrant	48
pt_line_distance	104
get_distance	Inlined
get_angle	Inlined
get_slope	Inlined
sqrt	Optimized out
DOUBLECOMPARE	Inlined
fabs	Optimized out
pow	Optimized out

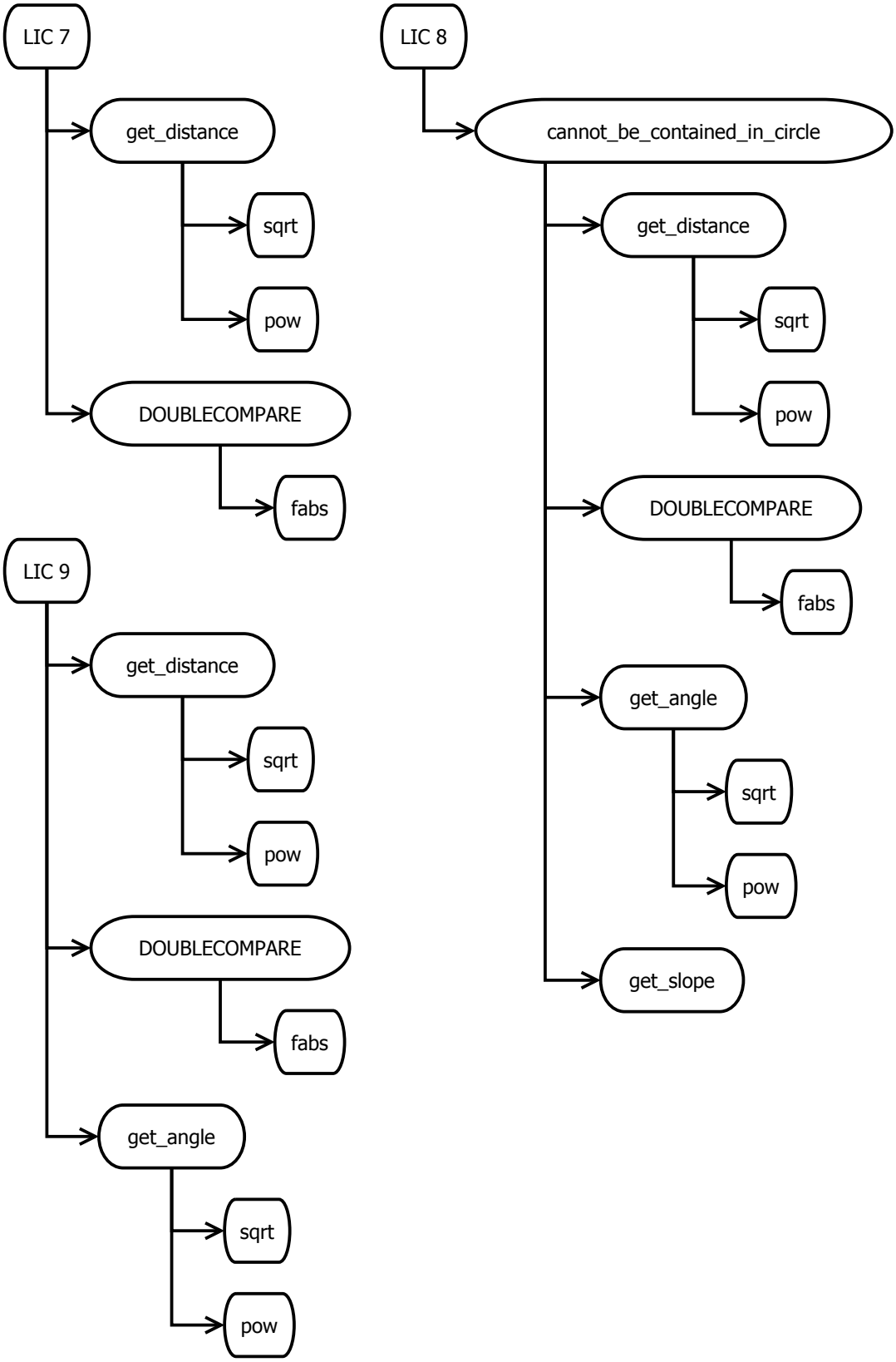
The worst case stack usage is when either LIC 8, or LIC 13 are called. The stack depth in these cases is 920 bytes. This is with the following compilation flags: *-ffloat - store -O2 - Werror - Wextra - Wall*

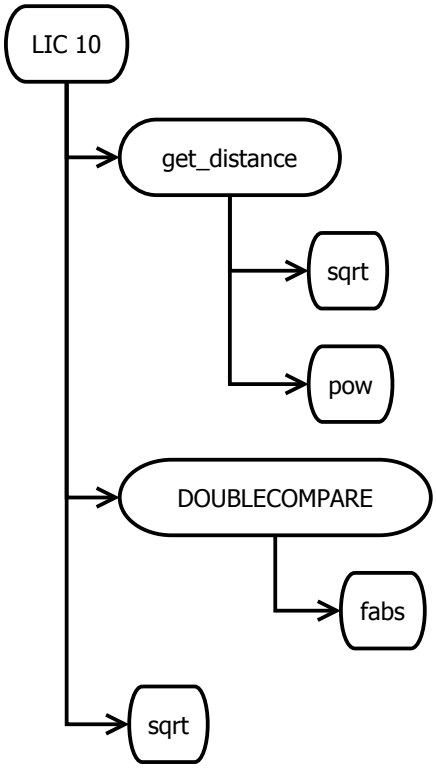
Program Callgraphs

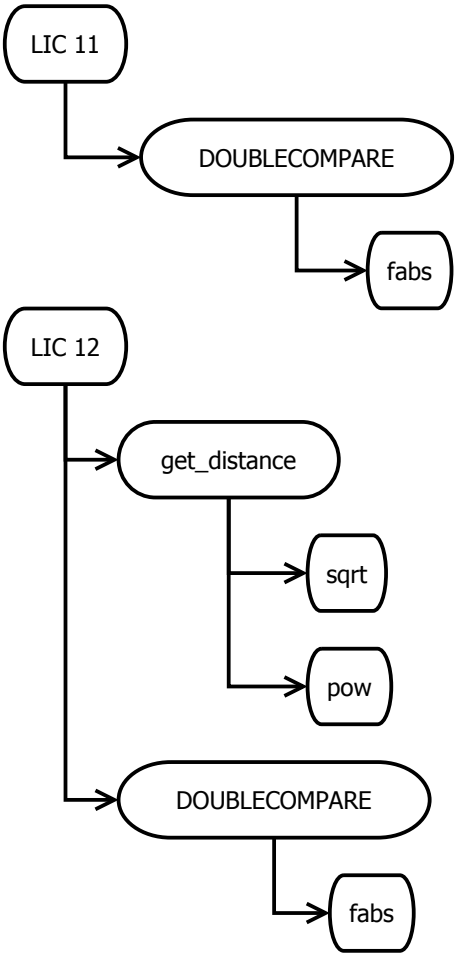
The following are diagrams showing the progressive nesting of the C-functions called by our implementation of the DECIDE function. Function calls aligned vertically on the same column were called in sequential order. The actual callgraph for the DECIDE function isn't shown due to its size. Each of the LIC functions are called in a sequential manner.

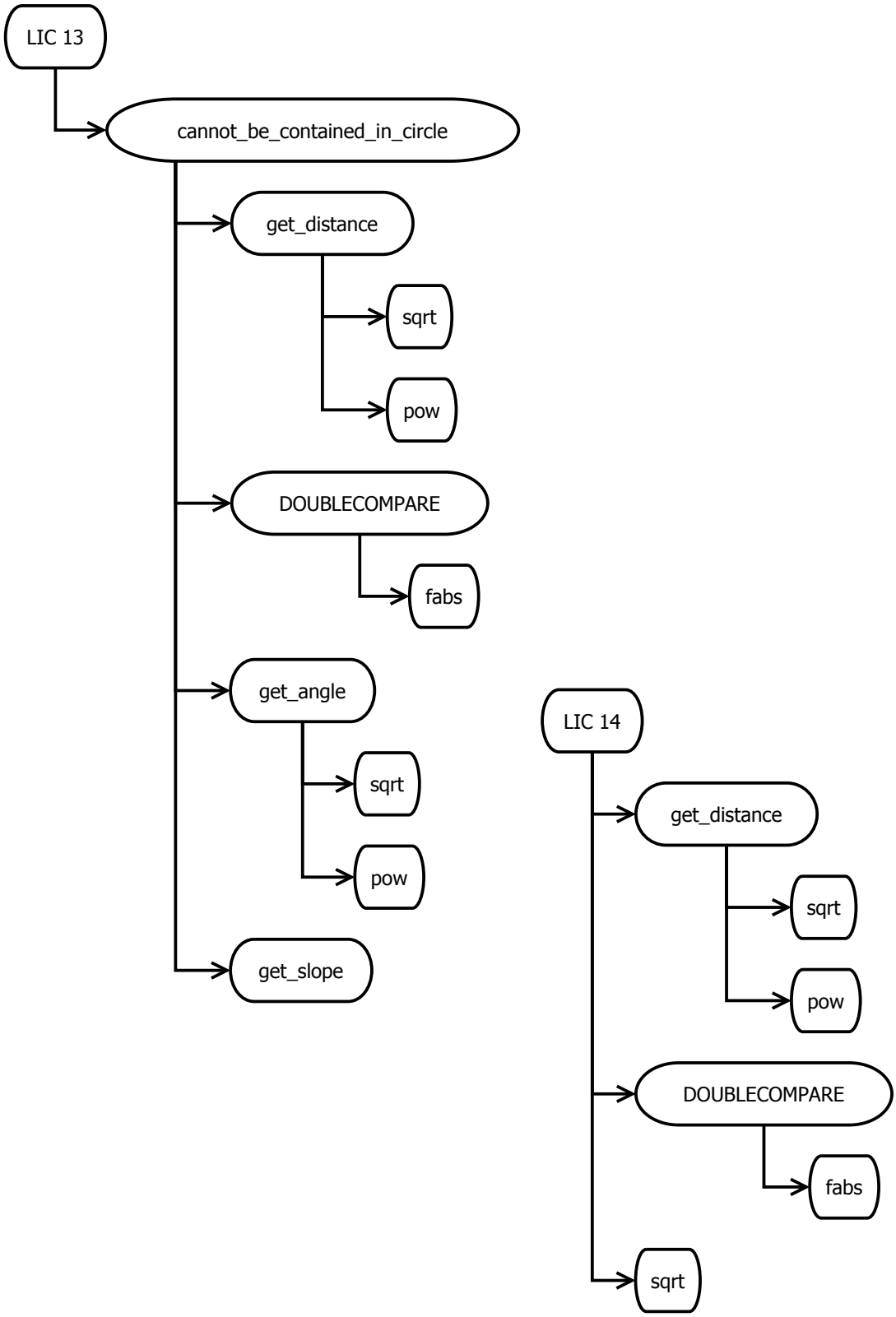

LIC 0
sqrt
DOUBLECOMPARE
fabs
LIC 1
cannot_be_contained_in_circle
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_angle
sqrt
pow
get_slope
LIC 2
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_angle
sqrt
pow


LIC 3
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
sqrt
LIC 4
get_quadrant
DOUBLECOMPARE
fabs
LIC 5
DOUBLECOMPARE
fabs
LIC 6
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_slope
pt_line_distance
sqrt
DOUBLECOMPARE
fabs


LIC 7
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
LIC 8
cannot_be_contained_in_circle
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_angle
sqrt
pow
get_slope
LIC 9
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_angle
sqrt
pow


LIC 10
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
sqrt


LIC 11
DOUBLECOMPARE
fabs
LIC 12
get_distance
sqrt
pow
DOUBLECOMPARE
fabs


LIC 13
cannot_be_contained_in_circle
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
get_angle
sqrt
pow
get_slope
LIC 14
get_distance
sqrt
pow
DOUBLECOMPARE
fabs
sqrt