LAB3

TODOS LOS MAKES SON DESDE EL DOCKER PARA COMPILAR EN ARM Y LUEGO COPIADOS A LA ZEDBOARD

1 Previous Work

1.1 Arithmetic Expression Optimizations and Routine Specializations

1)

__udivdi3 => división de una word sin signo por otra
__umoddi3 => resto en la división de una word sin signo por otra

1.2 Memoization

2)

No, no hay instrucciones que implementen las rutinas sin() y cos() para Cortex A9, hay como una extendida que te permite utilizar unas instrucciones sin() y cos().

2 Arithmetic Expression Optimizations and Bit Compression

3.a)

$ *usr*/bin/time -o time0 -p ./primers.0
$ *usr*/bin/time -o time01 -p ./primers.0
$ *usr*/bin/time -o time02 -p ./primers.0
$ *usr*/bin/time -o time03 -p ./primers.0
$ *usr*/bin/time -o time04 -p ./primers.0
De media unos 32,67 s de tiempo real

3.b)

$ valgrind --tool=callgrind ./primers.g0 100000
Esto me genera callgrind.out.1457 que lo copio al disco duro (ompss)
Y ejecuto lo siguiente para analizar el comportamiento de primers.c
$ kcachegrind callgrind.out.1457

3.c)

__udivmoddi4 es la función que más consume con un 76,36%
Hay dos operaciones en clearBit y getBit donde es llamada la función __udivmoddi4 que se pueden mejorar haciendo operaciones bit a bit
… (bitSS / ba→bitsPerInt);
Mejora: (bitSS >> 2);
… (bitSS % ba→bitsPerInt);
Mejora: (bitSS & 0x00000003);

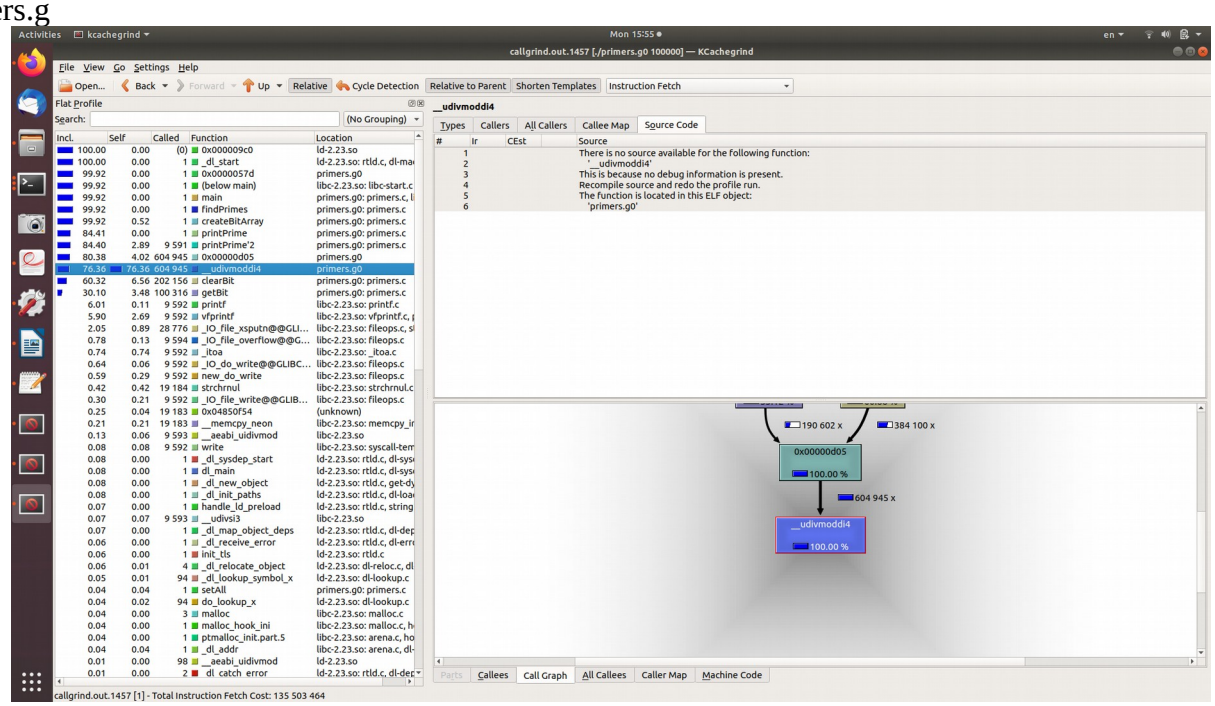
3.d)

$ *usr*/bin/time -o time3 -p ./primers.3
$ *usr*/bin/time -o time31 -p ./primers.3
$ *usr*/bin/time -o time32 -p ./primers.3
$ *usr*/bin/time -o time33 -p ./primers.3
$ *usr*/bin/time -o time34 -p ./primers.3
De media unos 7,15 s de tiempo real

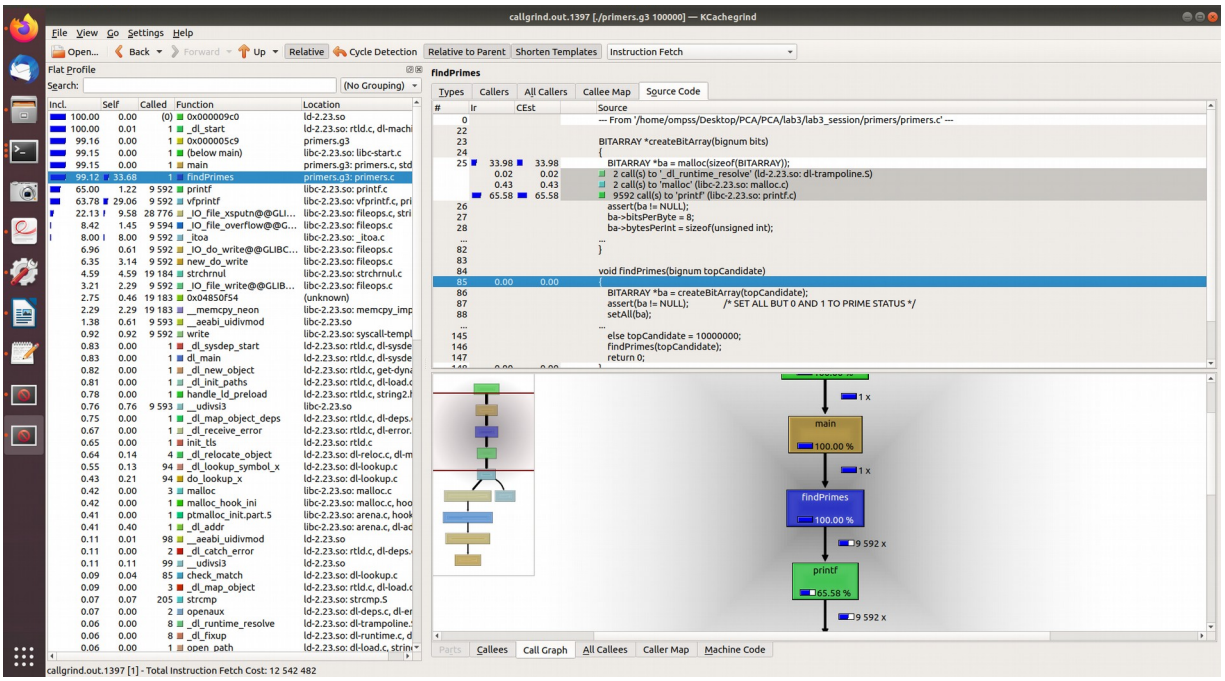
3.e)

SpeedUp de 4,57
$ valgrind –tool= callgrind ./primers.g3 100000
Genera un archivo callgrind.out.1397
Miro los ciclos estimados y el número de llamadas
$ callgrind_annotate --auto=yes callgrind.out.1457 > callgrind_annotate.1457
$ callgrind_annotate --auto=yes callgrind.out.1397 > callgrind_annotate.1397
Utilizo kcachegrind para observar con más detalle el comportamiento de primers.g3
$ kcachegrind callgrind.out.1397
Observo que ahora varias llamadas a algunas funciones han desaparecido en -O3 además

primers.g



primers.g3



4.a)

$ *usr*/bin/time -o time0 -p ./pi.0 5000
$ *usr*/bin/time -o time1 -p ./pi.0 5000
$ *usr*/bin/time -o time2 -p ./pi.0 5000
$ *usr*/bin/time -o time3 -p ./pi.0 5000
$ *usr*/bin/time -o time4 -p ./pi.0 5000
De media unos 14,78 s

$ *usr*/bin/time -o time30 -p ./pi.3 5000
$ *usr*/bin/time -o time31 -p ./pi.3 5000
$ *usr*/bin/time -o time32 -p ./pi.3 5000
$ *usr*/bin/time -o time33 -p ./pi.3 5000
$ *usr*/bin/time -o time34 -p ./pi.3 5000
De media unos 6,02 s

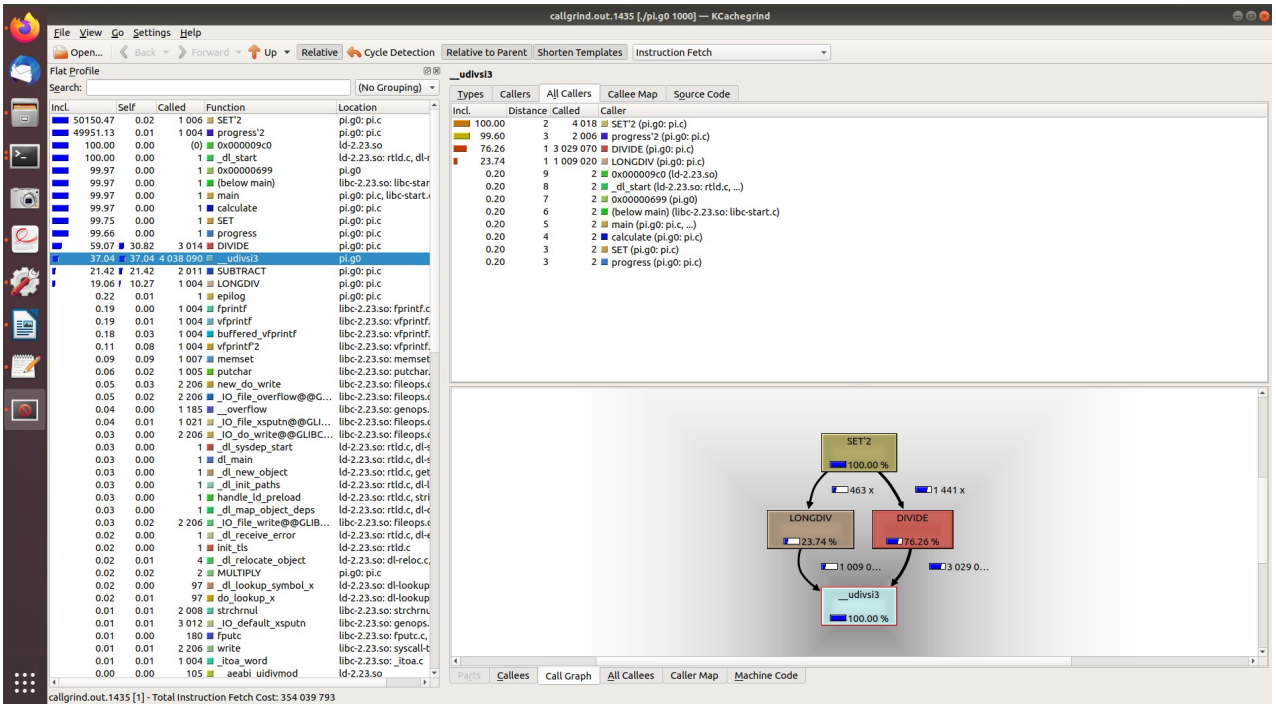
SpeedUp de 2,46

4.b)

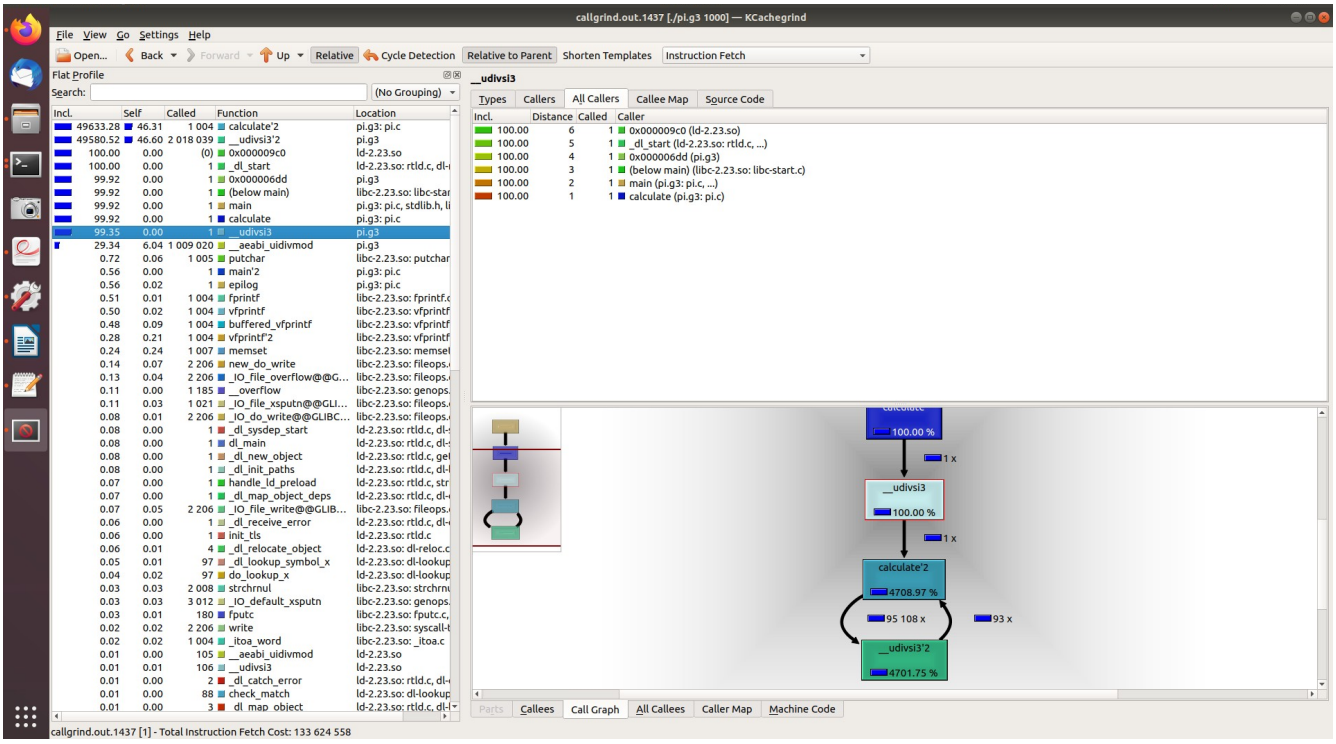
$ valgrind --tool=callgrind ./pi.g0 1000
$ valgrind --tool=callgrind ./pi.g3 1000
$ kcachegrind callgrind.out.1435
$ kcachegrind callgrind.out.1437

Esto me genera callgrind.out.1435 y 1437 que lo copio al disco duro (ompss) .Y ejecuto kcachegrind para ver el comportamiento de los ejecutables. Observo que para pi.g0 hay un 99,75 % de uso para la función calculate que llama a SET y esta a LONGDIV y DIVIDE las cuales llaman muchas veces a __udivsi3 que finalmente, es la que utiliza más la cpu. Para pi.g3 todas estan funciones desaparecen y calculate llama unicamente a __udivsi3, seguramente porque habrá hecho inline de estas funciones ahorrando llamadas a estas. Además, para LONGDIV y DIVIDE se observa con "**objdump -d**" que en vez de utilizar un push al principio para guardar unos registros, utiliza stm que guarda múltiples registros, ahorrando así la reserva posterior de estos.

pi.g



pi.g3



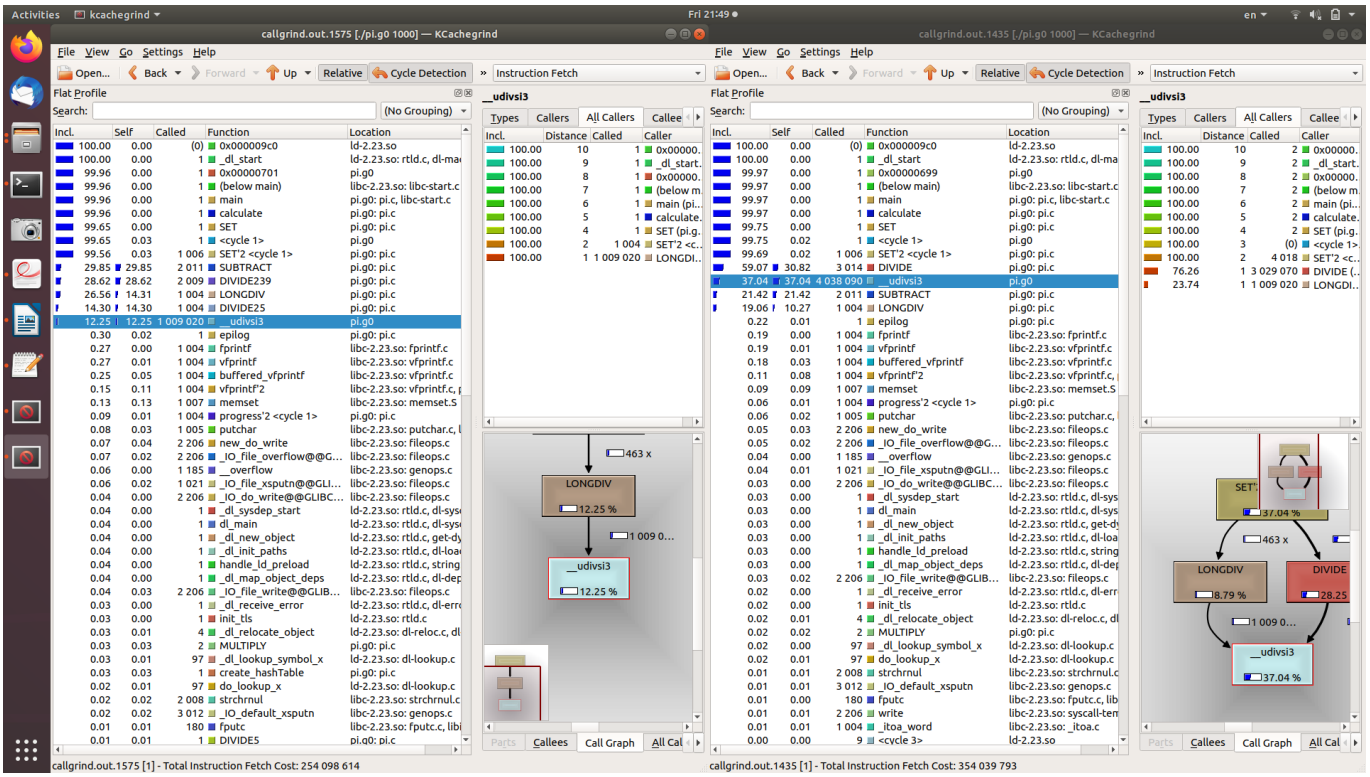
3 Routine Specializations

5)

He definido una hash table al ver que los resultados para una determinada u y n, los valores r y x[k] (que son los que nos interesan), siempre son los mismos. Y para cada n (5,25,239), hay una función diferente. Para hacerlo más fácil.

6)

$ valgrind --tool=callgrind ./pi.g0 1000 //pi.c mio, el de piv2
$ kcachegrind callgrind.out.1435
$ kcachegrind callgrind.out.1575
Vemos que hacemos menos llamadas a __udivsi3 de 4 millones a 1 millon más o menos



7)

$ *usr*/bin/time -o time00 -p ./pi.0 5000
$ *usr*/bin/time -o time01 -p ./pi.0 5000
$ *usr*/bin/time -o time02 -p ./pi.0 5000
$ *usr*/bin/time -o time03 -p ./pi.0 5000
$ *usr*/bin/time -o time04 -p ./pi.0 5000
De media unos 12,55 s
Speedup de 1,18, muy poco, pero hemos probado otras cosas y este es nuestro mejor y humilde resultado

Buffering

8.b y 8.c)

He analizado los dos programas, midiendo su tiempo de ejecución, utilizando valgrind para observar su comportamiento y no he visto diferencia alguna.

8.d)

Con strace -c ./trigon.pg3 1000 nos da lo siguiente

99.74	0.398997	0	2003335		3334 write
0.26	0.001059	0	8730		rt_sigreturn
0.00	0.000000	0	1		open
0.00	0.000000	0	1		close
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		1 access
0.00	0.000000	0	4		brk
0.00	0.000000	0	1		readlink
0.00	0.000000	0	1		munmap
0.00	0.000000	0	2		setitimer
0.00	0.000000	0	1		uname
0.00	0.000000	0	2		writev
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		mmap2
0.00	0.000000	0	1		set_tls
100.00	0.400056		2012084		3335 total

Es decir, write tarda 0.398997 s como elapsed time con 2003335 llamadas

8.e)

Hemos eliminado un bucle evitandonos a si una comparación, más un incremento n veces. Ahora solo hacemos una comparación y un solo incremento. Además hemos eliminado una escritura a memoria haciendo que rot sea un vector de 10 y poniendo a en la posición 0

8.f)

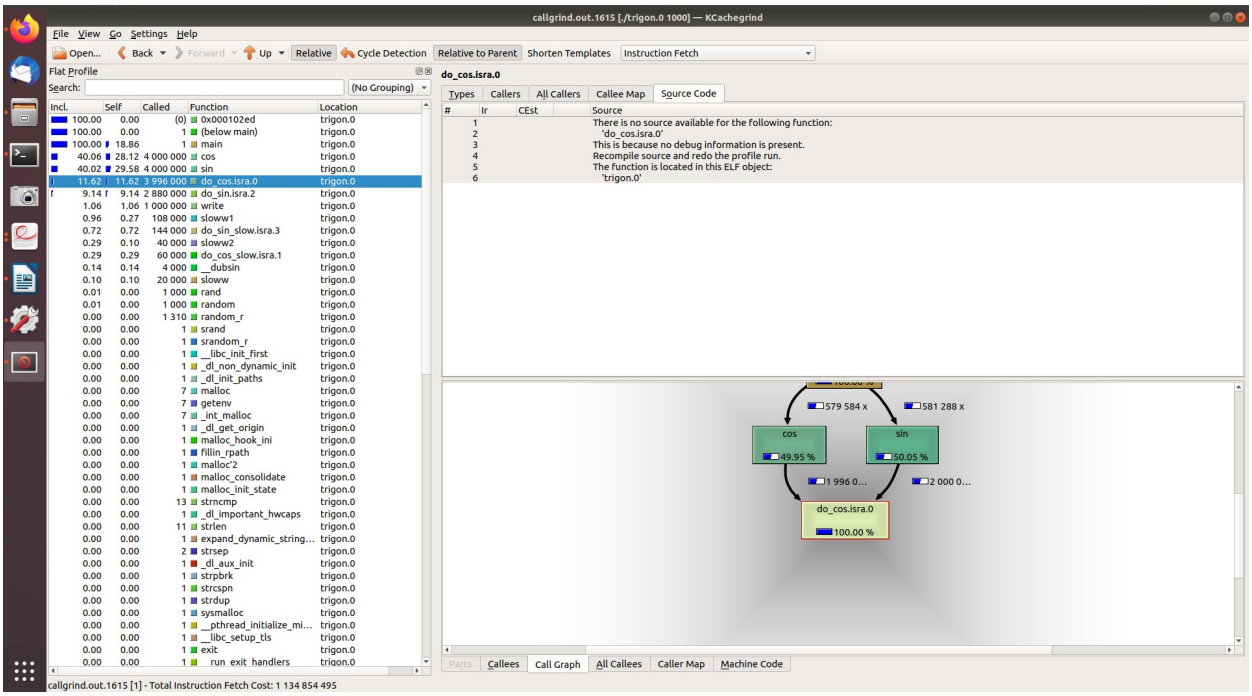
He utilizado time para saber los tiempos de la versión original y nuestra versión y nos da que el original de media tarda 26,35 s y nuestra versión 15,02 s . Para los dos casos hemos utilizado un argumento de 1000 para que fuese corta la ejecución. El speedup resultante es de 1,75

5 Memoization

9)

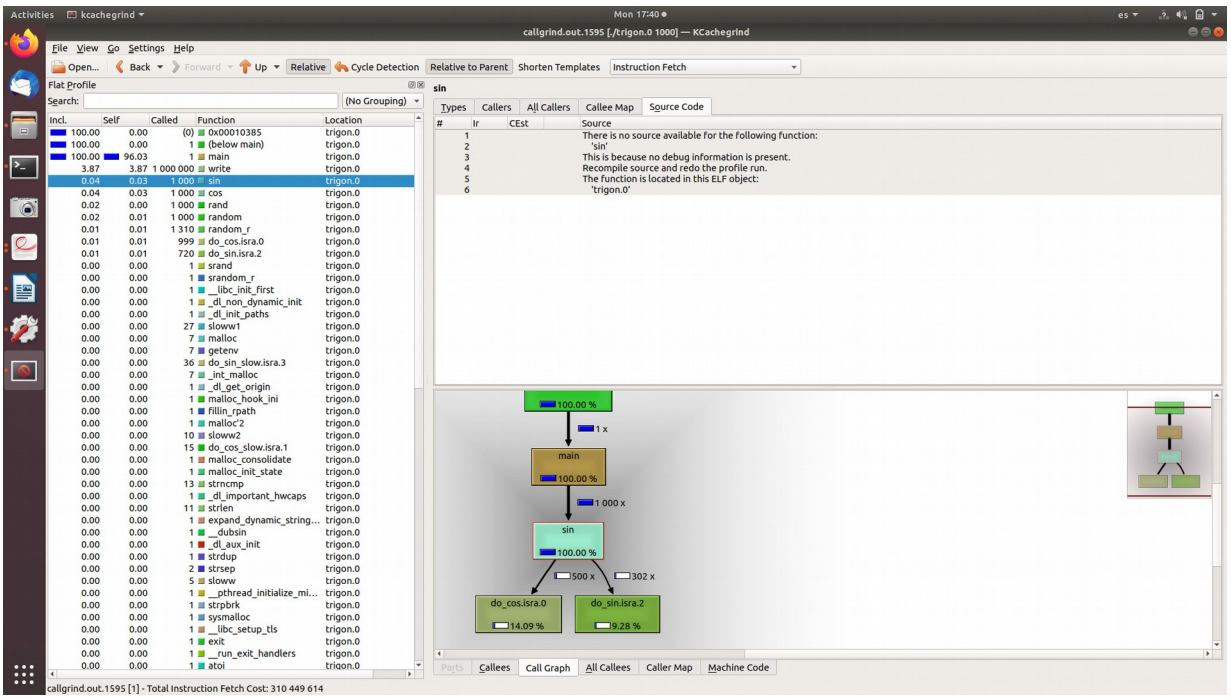
Sobre él mismo al componer un 20,76 % del programa las funciones sin() y cos(), el speedup sería de 1,26 y sobre el original un 2,03

trigon v2



11 y 12)

trigon memoization



Como observamos hemos reducido las llamadas a las funciones de sin() y cos() bastante. De hecho, al calcular el tiempo medio nos sale un 12,12 s , que respecto al original es un 2 de speedup, casi el máximo que se puede conseguir.