WCET Analysis Lab: Assignment 1

Markus Klein
Johannes Kasberger

SS 2012

Problem 1
Recommended (3): As a warm-up exercise, follow the instruction in Timing Analysis Lab: First Steps
Q: How long does it take to execute simple once, according to measurements, and according to the static analysis?

Answer

- Measurements: 763 incl. function call - 70 overhead = 693 cycles
- Static analysis: 705 cycles

We used the compiler flag -O1.

The difference is 12 cycles so the measurement is not far away from the static analysis. After one run we measured 1018 cycles but after reflashing the target the result was 763 cycles again.

Problem 2
Recommended (3): Also extract the instruction trace as outlined in Timing Analysis Lab: First Steps. Then compare the number of cycles needed in one iteration of the loop, with the number of cycles aiT calculated.
Q: Do they coincide? What is the total number of cycles needed to execute simple according to the instruction trace buffer?

Answer
To number of cycles for one iteration we measured 77 cycles. The static analysis results in 77 cycles.

```
10162210   40001210   call   0x400011a0              [40001210]
10162233   40001214   st   %g1, [%l1]                [40000000 07735935]
10162243   400011a0   mov   42, %g3                  [0000002a]
10162253   400011a4   mov   0, %g2                   [00000000]
1+2+1+1 = 5

10162263   400011a8   add   %g3, 1, %g1              [0000002b]
```

```
10162273   400011ac   add   %g1,  1,  %g1              [0000002c]
10162283   400011b0   add   %g1,  1,  %g1              [0000002d]
10162293   400011b4   add   %g1,  1,  %g1              [0000002e]
10162303   400011b8   add   %g1,  1,  %g1              [0000002f]
10162313   400011bc   add   %g1,  1,  %g1              [00000030]
10162323   400011c0   add   %g1,  1,  %g1              [00000031]
10162333   400011c4   add   %g1,  1,  %g3              [00000032]
10162343   400011c8   add   %g2,  1,  %g2              [00000001]
10162353   400011cc   cmp   %g2,  8                    [fffffff9]
10162363   400011d0   bne   0x400011a8                 [4000125c]
10162373   400011d4   nop                              [00000000]
14

10162383   400011a8   add   %g3,  1,  %g1              [00000033]
10162393   400011ac   add   %g1,  1,  %g1              [00000034]
10162404   400011b0   add   %g1,  1,  %g1              [00000035]
10162414   400011b4   add   %g1,  1,  %g1              [00000036]
10162424   400011b8   add   %g1,  1,  %g1              [00000037]
10162434   400011bc   add   %g1,  1,  %g1              [00000038]
10162444   400011c0   add   %g1,  1,  %g1              [00000039]
10162454   400011c4   add   %g1,  1,  %g3              [0000003a]
10162464   400011c8   add   %g2,  1,  %g2              [00000002]
10162474   400011cc   cmp   %g2,  8                    [fffffffa]
10162484   400011d0   bne   0x400011a8                 [4000125c]
10162494   400011d4   nop                              [00000000]
10162504   400011a8   add   %g3,  1,  %g1              [0000003b]
10162514   400011ac   add   %g1,  1,  %g1              [0000003c]
10162524   400011b0   add   %g1,  1,  %g1              [0000003d]
10162534   400011b4   add   %g1,  1,  %g1              [0000003e]
10162544   400011b8   add   %g1,  1,  %g1              [0000003f]
10162554   400011bc   add   %g1,  1,  %g1              [00000040]
10162564   400011c0   add   %g1,  1,  %g1              [00000041]
10162574   400011c4   add   %g1,  1,  %g3              [00000042]
10162584   400011c8   add   %g2,  1,  %g2              [00000003]
10162594   400011cc   cmp   %g2,  8                    [fffffffb]
10162604   400011d0   bne   0x400011a8                 [4000125c]
10162614   400011d4   nop                              [00000000]
10162624   400011a8   add   %g3,  1,  %g1              [00000043]
10162634   400011ac   add   %g1,  1,  %g1              [00000044]
10162644   400011b0   add   %g1,  1,  %g1              [00000045]
10162654   400011b4   add   %g1,  1,  %g1              [00000046]
10162664   400011b8   add   %g1,  1,  %g1              [00000047]
10162674   400011bc   add   %g1,  1,  %g1              [00000048]
10162684   400011c0   add   %g1,  1,  %g1              [00000049]
10162694   400011c4   add   %g1,  1,  %g3              [0000004a]
10162704   400011c8   add   %g2,  1,  %g2              [00000004]
```

```
10162714   400011cc   cmp   %g2, 8                      [fffffffc]
10162724   400011d0   bne   0x400011a8                  [4000125c]
10162734   400011d4   nop                               [00000000]
10162744   400011a8   add   %g3, 1, %g1                 [0000004b]
10162754   400011ac   add   %g1, 1, %g1                 [0000004c]
10162764   400011b0   add   %g1, 1, %g1                 [0000004d]
10162774   400011b4   add   %g1, 1, %g1                 [0000004e]
10162784   400011b8   add   %g1, 1, %g1                 [0000004f]
10162795   400011bc   add   %g1, 1, %g1                 [00000050]
10162805   400011c0   add   %g1, 1, %g1                 [00000051]
10162815   400011c4   add   %g1, 1, %g3                 [00000052]
10162825   400011c8   add   %g2, 1, %g2                 [00000005]
10162835   400011cc   cmp   %g2, 8                      [fffffffd]
10162845   400011d0   bne   0x400011a8                  [4000125c]
10162855   400011d4   nop                               [00000000]
10162865   400011a8   add   %g3, 1, %g1                 [00000053]
10162875   400011ac   add   %g1, 1, %g1                 [00000054]
10162885   400011b0   add   %g1, 1, %g1                 [00000055]
10162895   400011b4   add   %g1, 1, %g1                 [00000056]
10162905   400011b8   add   %g1, 1, %g1                 [00000057]
10162915   400011bc   add   %g1, 1, %g1                 [00000058]
10162925   400011c0   add   %g1, 1, %g1                 [00000059]
10162935   400011c4   add   %g1, 1, %g3                 [0000005a]
10162945   400011c8   add   %g2, 1, %g2                 [00000006]
10162955   400011cc   cmp   %g2, 8                      [fffffffe]
10162965   400011d0   bne   0x400011a8                  [4000125c]
10162975   400011d4   nop                               [00000000]
10162985   400011a8   add   %g3, 1, %g1                 [0000005b]
10162995   400011ac   add   %g1, 1, %g1                 [0000005c]
10163005   400011b0   add   %g1, 1, %g1                 [0000005d]
10163015   400011b4   add   %g1, 1, %g1                 [0000005e]
10163025   400011b8   add   %g1, 1, %g1                 [0000005f]
10163035   400011bc   add   %g1, 1, %g1                 [00000060]
10163045   400011c0   add   %g1, 1, %g1                 [00000061]
10163055   400011c4   add   %g1, 1, %g3                 [00000062]
10163065   400011c8   add   %g2, 1, %g2                 [00000007]
10163075   400011cc   cmp   %g2, 8                      [ffffffff]
10163085   400011d0   bne   0x400011a8                  [4000125c]
10163095   400011d4   nop                               [00000000]
10163105   400011a8   add   %g3, 1, %g1                 [00000063]
10163115   400011ac   add   %g1, 1, %g1                 [00000064]
10163125   400011b0   add   %g1, 1, %g1                 [00000065]
10163135   400011b4   add   %g1, 1, %g1                 [00000066]
10163145   400011b8   add   %g1, 1, %g1                 [00000067]
10163155   400011bc   add   %g1, 1, %g1                 [00000068]
10163165   400011c0   add   %g1, 1, %g1                 [00000069]
```

```
10163175   400011c4   add   %g1 ,  1 ,  %g3              [0000006a]
10163186   400011c8   add   %g2 ,  1 ,  %g2              [00000008]
10163196   400011cc   cmp   %g2 ,  8                     [00000000]
10163206   400011d0   bne   0x400011a8                   [4000125c]
10163216   400011d4   nop                                [00000000]
8*14
10163228   400011d8   retl                               [400011d8]
1
```

5+8∗14+1

Just based on the number of instructions we calculated the number of cycles needed = 118 cycles. This is much shorter than the result of the static analysis. The reason for this is that we don't know the exact memory timing and the instruction decode takes more time.

Problem 3

Mandatory (4): First, create a project containing the files contained in the insertion sort folder of the task specification. Now complete the function main.c:run(), executing insertion sort a few times, with array size 32 and different input data. Measure the minimum and maximum time needed to execute the sort function.

Q: What were the results of the measurement? How many test sets would do you need to cover all possible execution path?

Answer

We used the compiler flag -Os.

Case	**measurement**[*cycles*]
Best-case (pre sorted)	3659
Worst-case (upside down sorted)	44267
Average-case (unsorted)	24875

With the test data $\{5, 3, 4\}$ all possible paths in insertion sort function are entered once.

Problem 4

Mandatory (5): Add loop bounds and additional flow facts for insertion sort.c:insertion sort(), using the symbolic name @size for the size of the array to be sorted. Next, analyze the WCET of insertion sort, assuming an array size of 32. Keep the array size as a symbolic name (user register @size). Finally, write a test function which calls insertion sort more than once, with different array sizes (e.g., 16,32 and 64). Also repeat the static analysis with different array sizes.

Q: How many cycles do you need to execute insertion sort according to the static analysis?

Answer

We tried to analyse sort.c:insertion sort() directly but a3 wasn't able to compute the WCET. It did not accept our loop bounds. When we added a simple function that called sort.c:insertion sort() the result was 77224 cycles.

Q: What results do you get for an array size of 8,16 or 64, using measurements and static analysis?

Answer

method	**size 16**[$cycles$]	**size 32**[$cycles$]	**size 64**[$cycles$]
measurement	11527	34579	132899
static	18232	77224	318088

Q: In addition to the size of the array, what other aspects of the input data might influence the WCET?

Answer

The structure of the data. For instance: The worst-case occurs when the input data is sorted upside down. The best-case occurs if the input data is already sorted.

Problem 5

Recommended (8 Points): Assume that your goal was to find out the WCET of task.c:task(). Before analyzing the execution time, you should answer a few basic questions about the input data for the monitoring task, and analyze the control flow on the source code level.
Q: What is the set of input data which might influence the execution time of the task at the software side? Is it tractable to enumerate every possible input? Which loops need to be bounded? Add all loop bounds and flow facts you can find to the file task.c (as source code annotations).

Answer

Problem 6

Recommended (8 Points): Analyze the fft() function called in task.c. Try to find loop bounds for the Fast Fourier Transform implementation (fixedpoint.c:fp radix2fft withscaling) first. If you have difficulties finding them, add a debug statement and run the transform with different input data sizes. Add flow constraints relating the execution frequency of the inner loops with the functions execution frequency. Finally, try to analyze the execution time using aiT. There is already a timing measurement for the fft in the executable, so it is easy to compare the number of cycles estimated to execute the function.
Q: Compare the worst-case number of iterations for the inner loop with and without using these flow constraints. Finally, think about the complexity of calculating loop bounds for FFT.

Answer

Q: Does the FFT loop bound depend on the input data?

Answer

Problem 7

Optional Challenge (5 Bonus Points): Try to analyze the WCET of task.c:task() using aiT. If you attempt to solve this challenge, use the control flow graph and disassembling capabilities of aiT, and be sure to understand the source code you are analyzing.

Problem 8

Mandatory (4): Answer the following questions

Q: How much time did you spend writing annotations and analyzing the code? Was it less or more than you expected? How much time did you spend on this first assignment?

Answer

Q: What is the ratio between observed and actual execution time? Discuss the causes of the overestimation.

Answer

Q: As you learned, sometimes it is necessary to annotate the assembler code. Why? What problems can you see because of this?

Answer