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
We used the compiler flag -O1.

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

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. We can't explain this difference.

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

```
time       address    instruction                result
10839553   40001210   call   0x400011a0          [40001210]
10839568   40001214   st   %g1, [%l1]            [40000000 07735939]
10839575   400011a0   mov   42, %g3              [0000002a]
10839582   400011a4   mov   0, %g2               [00000000]
10839589   400011a8   add   %g3, 1, %g1         [0000002b]
10839596   400011ac   add   %g1, 1, %g1         [0000002c]
```

```
10839603   400011b0   add  %g1, 1, %g1          [0000002d]
10839610   400011b4   add  %g1, 1, %g1          [0000002e]
10839617   400011b8   add  %g1, 1, %g1          [0000002f]
10839624   400011bc   add  %g1, 1, %g1          [00000030]
10839631   400011c0   add  %g1, 1, %g1          [00000031]
10839638   400011c4   add  %g1, 1, %g3          [00000032]
10839645   400011c8   add  %g2, 1, %g2          [00000001]
10839652   400011cc   cmp  %g2, 8               [fffffff9]
10839659   400011d0   bne  0x400011a8           [4000125c]
10839666   400011d4   nop                       [00000000]
10839673   400011a8   add  %g3, 1, %g1          [00000033]
10839680   400011ac   add  %g1, 1, %g1          [00000034]
10839687   400011b0   add  %g1, 1, %g1          [00000035]
10839694   400011b4   add  %g1, 1, %g1          [00000036]
10839701   400011b8   add  %g1, 1, %g1          [00000037]
10839708   400011bc   add  %g1, 1, %g1          [00000038]
10839715   400011c0   add  %g1, 1, %g1          [00000039]
10839722   400011c4   add  %g1, 1, %g3          [0000003a]
10839729   400011c8   add  %g2, 1, %g2          [00000002]
10839736   400011cc   cmp  %g2, 8               [fffffffa]
10839743   400011d0   bne  0x400011a8           [4000125c]
10839750   400011d4   nop                       [00000000]
10839757   400011a8   add  %g3, 1, %g1          [0000003b]
10839764   400011ac   add  %g1, 1, %g1          [0000003c]
10839771   400011b0   add  %g1, 1, %g1          [0000003d]
10839778   400011b4   add  %g1, 1, %g1          [0000003e]
10839785   400011b8   add  %g1, 1, %g1          [0000003f]
10839792   400011bc   add  %g1, 1, %g1          [00000040]
10839799   400011c0   add  %g1, 1, %g1          [00000041]
10839806   400011c4   add  %g1, 1, %g3          [00000042]
10839813   400011c8   add  %g2, 1, %g2          [00000003]
10839820   400011cc   cmp  %g2, 8               [fffffffb]
10839827   400011d0   bne  0x400011a8           [4000125c]
10839834   400011d4   nop                       [00000000]
10839841   400011a8   add  %g3, 1, %g1          [00000043]
10839848   400011ac   add  %g1, 1, %g1          [00000044]
10839855   400011b0   add  %g1, 1, %g1          [00000045]
10839862   400011b4   add  %g1, 1, %g1          [00000046]
10839869   400011b8   add  %g1, 1, %g1          [00000047]
10839876   400011bc   add  %g1, 1, %g1          [00000048]
10839883   400011c0   add  %g1, 1, %g1          [00000049]
10839890   400011c4   add  %g1, 1, %g3          [0000004a]
10839897   400011c8   add  %g2, 1, %g2          [00000004]
10839904   400011cc   cmp  %g2, 8               [fffffffc]
10839911   400011d0   bne  0x400011a8           [4000125c]
10839918   400011d4   nop                       [00000000]
```

```
10839925   400011a8   add  %g3, 1, %g1          [0000004b]
10839932   400011ac   add  %g1, 1, %g1          [0000004c]
10839939   400011b0   add  %g1, 1, %g1          [0000004d]
10839946   400011b4   add  %g1, 1, %g1          [0000004e]
10839953   400011b8   add  %g1, 1, %g1          [0000004f]
10839960   400011bc   add  %g1, 1, %g1          [00000050]
10839967   400011c0   add  %g1, 1, %g1          [00000051]
10839974   400011c4   add  %g1, 1, %g3          [00000052]
10839981   400011c8   add  %g2, 1, %g2          [00000005]
10839988   400011cc   cmp  %g2, 8               [fffffffd]
10839995   400011d0   bne  0x400011a8           [4000125c]
10840002   400011d4   nop                       [00000000]
10840009   400011a8   add  %g3, 1, %g1          [00000053]
10840016   400011ac   add  %g1, 1, %g1          [00000054]
10840023   400011b0   add  %g1, 1, %g1          [00000055]
10840030   400011b4   add  %g1, 1, %g1          [00000056]
10840037   400011b8   add  %g1, 1, %g1          [00000057]
10840044   400011bc   add  %g1, 1, %g1          [00000058]
10840051   400011c0   add  %g1, 1, %g1          [00000059]
10840058   400011c4   add  %g1, 1, %g3          [0000005a]
10840065   400011c8   add  %g2, 1, %g2          [00000006]
10840072   400011cc   cmp  %g2, 8               [fffffffe]
10840079   400011d0   bne  0x400011a8           [4000125c]
10840086   400011d4   nop                       [00000000]
10840093   400011a8   add  %g3, 1, %g1          [0000005b]
10840100   400011ac   add  %g1, 1, %g1          [0000005c]
10840107   400011b0   add  %g1, 1, %g1          [0000005d]
10840114   400011b4   add  %g1, 1, %g1          [0000005e]
10840121   400011b8   add  %g1, 1, %g1          [0000005f]
10840128   400011bc   add  %g1, 1, %g1          [00000060]
10840135   400011c0   add  %g1, 1, %g1          [00000061]
10840142   400011c4   add  %g1, 1, %g3          [00000062]
10840149   400011c8   add  %g2, 1, %g2          [00000007]
10840156   400011cc   cmp  %g2, 8               [ffffffff]
10840163   400011d0   bne  0x400011a8           [4000125c]
10840170   400011d4   nop                       [00000000]
10840177   400011a8   add  %g3, 1, %g1          [00000063]
10840184   400011ac   add  %g1, 1, %g1          [00000064]
10840191   400011b0   add  %g1, 1, %g1          [00000065]
10840198   400011b4   add  %g1, 1, %g1          [00000066]
10840205   400011b8   add  %g1, 1, %g1          [00000067]
10840212   400011bc   add  %g1, 1, %g1          [00000068]
10840219   400011c0   add  %g1, 1, %g1          [00000069]
10840226   400011c4   add  %g1, 1, %g3          [0000006a]
10840233   400011c8   add  %g2, 1, %g2          [00000008]
10840240   400011cc   cmp  %g2, 8               [00000000]
```

```
10840247   400011d0   bne   0x400011a8                [4000125c]
10840254   400011d4   nop                             [00000000]
10840263   400011d8   retl                            [400011d8]

Difference: 710 cycles
```

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

To cover all possible execution paths to sort a array of 32 32 bit integers we need maximal $2^{32} * 32 = 137438953472$ test cases.

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

The static analysis resulted in 74694 cycles.

Q: What results do you get for an array size of 16, 32 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?

Answer
If many samples are missing the interpolation of the data has big influence on the runtime. Most of the other loops just depend on the amount of samples used for calculation but this does not depend on the input data and is determined at compile time.

Q: Is it tractable to enumerate every possible input?

Answer
No it's not traceable. The result of the interpolation is dependent on the history of the last samples so enumerating the history and the inputs would require many differnt testcases.

Q: Which loops need to be bounded?

Answer
The loops we found are in the merge samples function.

Q: Add all loop bounds and flow facts you can find to the file task.c (as source code annotations).

Answer

```
for(i = valid+1;
    i < cnt;/* ai: loop here MAX (@inputcount+4); */
    i++)
{

      if(i >= 0)
      {
              x = xs[i];
              /* ai: flow (here) <= 32 ("merge_samples"); */
              sample_buffer_set(sbuf,i,x);
      }
      else
      {
              x = sample_buffer_get(sbuf,i);
              /* ai: flow (here) <= 4 ("merge_samples"); */
      }

  /* If the sample is not missing, interpolate the ones before if the range is accepta
      if(! IS_VALUE_MISSING(x))
      {
        /* Only interpolate if we interpolate at most MAX_CONSECUTIVE_MISSING samples
        int missing_samples = i - valid - 1;
        if(missing_samples > 0 && missing_samples <= MAX_CONSECUTIVE_MISSING)
        {

              for(j = i-1;j > valid;--j)
              {
            /* At most once for each invalid input sample */

                sample_value_t y = sample_buffer_get(sbuf,j);
                /* ai: loop here MAX 4; */
                /* ai: flow each (here) / ("merge_samples") is max 32; */
                if(! IS_VALUE_MISSING(y)) break;
                y = iinterpolate16(valid,sample_buffer_get(sbuf,valid),i,x,j);
                sample_buffer_set(sbuf,j, y);
              }
        }
        valid = i;
      }
}
```

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 with-scaling) 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
Comparison static analysis with measurement:

type	cycles
measurement	178807 - 124755
static with flow constraints	160344 - 159957

Comparison of inner loop in fp radix2fft withscaling with/without flow constraints:

type	cycles
static without flow constraints	1998624
static with flow constraints	79491

Without the flow constraint the inner loops have a big impact on WCET. The reason is that each inner loop is at most executed 32 times in each interation but this number changes with every iteration. We calculated the flow contraint with $\sum_{i=0}^{t-1} 2^i * 2^{t-i} = 192$ to represent this fact.

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

Answer
It just depends on the amount of the input data and not the data itself.

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

- Mandatory Part: 6 hours
- Recommended Part: 6 hours
- Protocol: 2 hours

We wored 14 hours on this assignment. It was more than expected, because we ran into many problems which were not directly related to the given exercises (e.g. configuration errors in aiT and so on).

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

Answer
For the static analysis we encountered a result that sometime was twice as long as our measurements. The causes of overestimation are too less restrictive loop bounds and flow annotations. The memory access times of the CPU model are a pessimistic estimation and can also be a source of overestimation.

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

Answer
The code has an other structure in assembler after the compiler has finished its optimisation. The source code annotation points to a line of code that is on another position in the assembler code. Sometimes the loop body is moved before the loop condition and then it's also hard to see where a source code annotation belongs to.

The problem of annotating the assember code is that every litte change and recompilation of a program can possibly result in totally different binaries and would require to adapt the assember annotations to this new binary.