Timing Analysis for Safety-Critical Real-Time Systems VU

Lab Report

Assignment 1

Konstantin Selyunin, Matr. Nr.1228206
e1228206@student.tuwien.ac.at
Vorname2 Nachname2, Matr. Nr.99012345
e99012345@student.tuwien.ac.at

April 17, 2014

Contents

1 Problem 1

1.1 Problem Statement

In directory `insertion_sort`, implement the function `main.c:run()`. This routine should call the insertion sort routine several times, with different arrays of size 32.

Q1: What is the minimum and maximum number of times the statement labeled `insertion_sort_move` might be executed for some input data (assuming an array size of N)?

Q2: How many test runs do you need to cover all possible paths through the function?

Q3: Report the minimum and maximum execution times measured for your test runs of insertion sort.

1.2 Solution

A1: For an array of size N the labeled statement `insertion_sort_move` might be executed from 0 (sorted) to $N \times (N-1)/2$ (reverse order) times (in our case for an array of 32 elements from 0 to 496).

A2: Assume that the length of the input array is N.

After an inspection of the sorting algorithm one can see that the algoritm behaviour does not depend on the exact values of the input array. It only depends on the relations between the array elements. The reason is that the array elements are only compared by relational operarors like ">" and the only operation invoked on array elements is copying. This means that control-flow path is the same for two arrays of lenght N with the same distribution of elements inversions.

In the very general case two arbitrary elements a and b of the input array can be related as $a < b$, $a = b$ or $a > b$, and the execution path might be influenced by this fact in three different ways. The total number of permutations, that take into account equalities and inversions, for arrays of length N can be expressed as

$$\sum_{n=1}^{N} \sum_{\substack{\bigwedge_{i=1}^{n} m_i \geq 1 \\ \sum_{i=1}^{n} m_i = N}} \binom{N}{m_1, m_2, \ldots, m_n}. \tag{1}$$

Potentially, each of these arrays might cause a sorting algorithm to follow a unique execution path. However, in the case of insertion sort algorithm we know that relations $a < b$ and $a = b$ cannot be distinguished. This reduces the number of paths accordint to (1) to the number of $N!$. A simpler explanation can be provided in this case. When the insertion sort algorithm processes the element at the position i ($1 \leq i \leq N$), all the elements at the previous positions $1 \ldots i-1$ are already sorted. For the element i there are now exactly i positions to be placed. This adds a factor i to the number of total pths that should be executed. Hence, we get for the total number of path an expression

$$\prod_{i=1}^{N} i = N!.$$

A3: The minimum execution time (in processor cycles) is $t_1 = 5477$ for the sorted array, the maximum is $t_2 = 30083$ for elements in reverse order as well as for an array of equal elements (e.g. all zeros). Sorting an array of random elements takes ‘usually’ 17/18k cycles. For details see Figure 2.

1.3 Listings

1.3.1 `main.c:run()`

```
/*
 * main.c
 *
 * Daniel Prokesch <daniel@vmars.tuwien.ac.at>, 2014
 * Benedikt Huber <benedikt@vmars.tuwien.ac.at>, 2010-2014
 *
 * WCET Analysis Lab
 * Cyber-Physical Systems Group
```

```c
 * Institute of Computer Engineering
 * Vienna University of Technology
 *
 * insertion sort - test driver
 */
#ifdef VERBOSE
#include <stdio.h>
#define V(x) x;
#else
#define V(x) do{}while(0)
#endif
#include "insertion_sort.h"
#include "measure.h"
#include <stdlib.h>


void run_tests()
{
  /* TODO: insert tests */
  const int input_size = 32;
  int input1 [input_size], input2 [input_size];
  int input3 [input_size], input4 [input_size];
  int input5 [input_size], input6 [input_size];
  int input7 [input_size];
  int i;

  for (i = 0; i <= input_size - 1; i++){
    input1[i] = i + 1;
    input2[i] = 32 - i;
    input3[i] = rand();
    input4[i] = rand();
    input5[i] = rand();
    input6[i] = 0;
    input7[i] = ~(1 << ((int) sizeof(int) * 8 - 1)) - i;
  }

  cycles_t t1, t2, t3, t4, t5, t6, t7;
  MEASUREMENT_START(t1);
  insertion_sort(input1, input_size);
  MEASUREMENT_STOP(t1);

  MEASUREMENT_START(t2);
  insertion_sort(input2, input_size);
  MEASUREMENT_STOP(t2);

  MEASUREMENT_START(t3);
  insertion_sort(input3, input_size);
  MEASUREMENT_STOP(t3);

  MEASUREMENT_START(t4);
  insertion_sort(input4, input_size);
  MEASUREMENT_STOP(t4);

  MEASUREMENT_START(t5);
  insertion_sort(input5, input_size);
  MEASUREMENT_STOP(t5);

  MEASUREMENT_START(t6);
  insertion_sort(input6, input_size);
  MEASUREMENT_STOP(t6);

  MEASUREMENT_START(t7);
  insertion_sort(input7, input_size);
  MEASUREMENT_STOP(t7);

  MEASUREMENT_DUMP(t1);
  MEASUREMENT_DUMP(t2);
  MEASUREMENT_DUMP(t3);
  MEASUREMENT_DUMP(t4);
  MEASUREMENT_DUMP(t5);
  MEASUREMENT_DUMP(t6);
  MEASUREMENT_DUMP(t7);

}

int main(int argc, char** argv)
{
  run_tests();
  return 0;
}
```

2 Problem 2

2.1 Problem Statement

Add loop bounds and additional flow facts for `insertion_sort.c:insertion_sort()`, and analyze the WCET of `insertion_sort`.

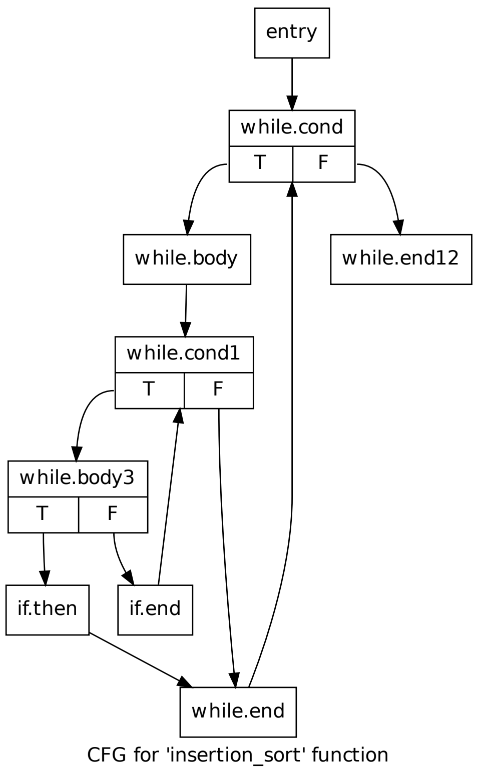
Q1: What is the WCET of insertion_sort, assuming the array has 8,16,32 or 64 elements? Compare the WCET for arrays of size 32 with your measurement results.

Q2: What other properties of the input (despite the size of the array) have a significant impact on the execution time? How could you take them into account during static WCET analysis?

Q3: Now try compiling with `-O1` (cf. the Makefile). Try to add your flow facts for array size 32. This might require a different method than source code annotations. List the maximally observed execution time and the WCET from analysis, and compare them to the results obtained with `-O0`.

Q4: (Bonus) Figure 1 depicts the *control-flow graph* (CFG) of the `insertion_sort` function as exported from the compiler intermediate representation. Formulate an ILP problem to calculate the WCET. That is, specify a set of (integer) variables, a linear objective function, and a set of linear constraints, such that the solution to the problem is an upper bound for the execution time of the program. The linear constraints consist of the structural constraints given by the CFG and the flow constraints you specified. For the basic block costs, assume a very simple model and use the costs given in Table **??**. Write an input file for an ILP solver[1], and check the solution calculated by the solver. Do not forget to give the number of times each basic block is executed in your solution.

[1]Recommendation: `http://sourceforge.net/projects/lpsolve`



Basic Block	Cost
entry	12
while.cond	6
while.body	15
while.cond1	4
while.body3	11
if.then	1
if.end	21
while.end	16
while.end12	1

Figure 1: Control-Flow Graph for `insertion_sort` (from LLVM IR) and costs for the execution of basic blocks.

2.2 Solution

A1: The results of WCET static analysis, obtained with `a3patmos` tool (column 2), are summarized in the table below:

Array size	WCET (cycles)	Measured
8	6512	-
16	22888	-
32	46904	30083
64	330808	-

For an array of 32 elements `a3patmos` gives result which is approximately 1.5 times larger then the largest measurement. If the 'triangulation' structure of the loops is not imposed as a constraint, the result is even more pessimistic (85.5k cycles for array size of 32).

A2: Number of inversions in the input array has significant impact on the execution time. In order to take it into account during static WCET analysis, we add linear constraint for maximum number of inversions in the inner loop.

A3: For this task we modify `Makefile` and include corresponding `insertion_sort_O1.*` targets. We also define annotations in a separate file (aiT.ais, see listing or supplement). The differences between execution times for the test cases and the WCET are summarized in the Figure 2. Optimizations performed by compiler reduce execution time by approximately factor of 3. The pitfall here is that the control flow graph from `a3patmos` tool does not give the valid addresses of instructions corresponding to `insertion_sort_move`: to solve this issue we obtain the addresses from the disassembly file (after analysing `insertion_sort` assembly code).

A4: In order to perform the analysis we build a directed graph that assigns execution time to edges but not vertices (Fig. 3). This is done by moving execution time from each block (corresponding to a vertex) to its outgoing edges. The same value (equal to the corresponding block execution cost) is assigned to all outgoing edges. Since we do not have outgoing edges form the block `while.end12`, we add one more vertex to the graph (which we call `exit`) and an edge between vertices `while.end12` and `exit`. The cost of `while.end12` block is assigned to the new edge.

A cost value and a number of executions is associated with each edge of the graph, such that the edge i has the cost t_i and the number of executions x_i correspondingly. The ILP problem consists of the objective function and the

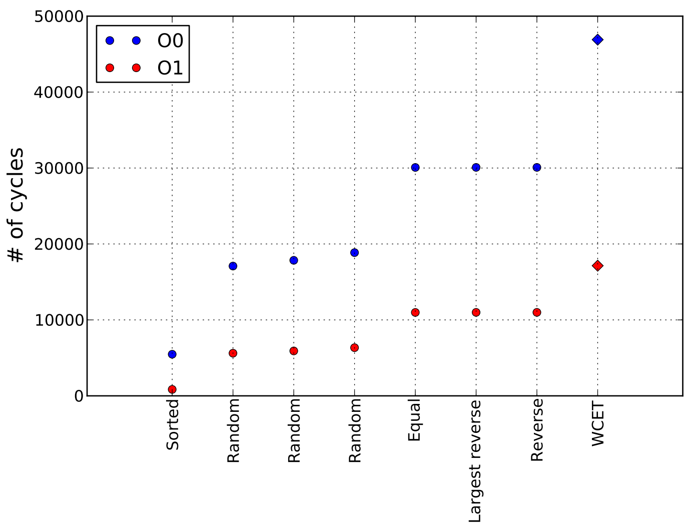
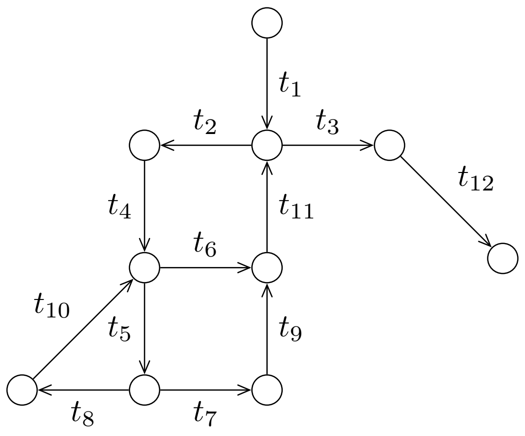


Figure 2: Execution time measurements and the WCET for `-O0` and `-O1`



Edge Cost	Basic Block
t_1	entry
$t_2 = t_3$	while.cond
t_4	while.body
$t_5 = t_6$	while.cond1
$t_7 = t_8$	while.body3
t_9	if.then
t_{10}	if.end
t_{11}	while.end
t_{12}	while.end12

Figure 3: Control-Flow Graph for ILP problem formulation

constraints. The objective function is maximization of the execution time:

$$\max_{x_1, x_2, \ldots x_{12}} \sum_{i=1}^{12} x_i t_i. \tag{2}$$

The constraints reflect the facts we know about control-flow of the function. The node constraints describe the fact that each program block is entered and left exactly the same number of times:

$$\begin{aligned} x_2 + x_3 &= x_1 + x_{11} \\ x_2 &= x_4 \\ x_4 + x_{10} &= x_5 + x_6 \\ x_5 &= x_7 + x_8 \\ x_7 &= x_9 \\ x_8 &= x_{10} \\ x_6 + x_9 &= x_{11} \\ x_3 &= x_{12} \end{aligned} \tag{3}$$

This control flow constraints itself cannot give a finite solution because of cycles in the graph. Thus, we add a requirement of a single function entrance and function exit or infinite looping:

$$\begin{aligned} x_1 &= 1 \\ x_{12} &\leq 1 \end{aligned} \tag{4}$$

This additional requirements still do not make the solution finite. We should add the constraints that describe loop bounds. We suppose that the length of the input array is N. From the previously done analysis we know that in this case the outer loop is executed exactly N times and the inner loop—not more than $N(N-1)/2$. Hence, we add

$$\begin{aligned} x_{11} &= N \\ x_{10} &\leq N(N-1)/2 \end{aligned} \tag{5}$$

Finally, there are physical feasibility constraints:

$$x_i \in \mathbb{N}, \quad \text{where } 1 \leq i \leq 12 \tag{6}$$

All the constraints (2–6) formulate the ILP problem for IPET analysis. Equation (6) should be passed to the solver through the type information, i.e. the solver will treat the variables as natural numbers implicitly.

The problem can be solved every time for each specific N and $t_1 \dots t_{12}$. However, the intuition and previous analysis say that the WCET should always correspond to the case when the inner loop is executed $N(N-1)/2$ times. Since the problem is not large, we can try to prove it analytically.

First of all, eliminate the redundant variables in node constraints:

$$\begin{aligned} x_2 + x_3 &= 1 + N \\ x_2 + x_8 &= x_5 + x_6 \\ x_5 &= x_7 + x_8 \\ x_6 + x_7 &= N \end{aligned} \tag{7}$$

and prove termination of the `insertion_sort` function. Termination is formulated as $x_{12} = 1$. From $x_3 = x_{12}$ and $x_{12} \leq 1$ we conclude $x_3 \leq 1$, i.e. we should prove that $x_3 = 1$ but not $x_3 = 0$.

Suppose that $x_3 = 0$. From (7) we have

$$\begin{cases} x_2 = 1 + N \\ x_2 + x_8 = x_5 + x_6 \\ x_5 = x_7 + x_8 \\ x_6 + x_7 = N \end{cases} \Rightarrow \begin{cases} 1 + N + x_8 = x_7 + x_8 + x_6 \\ x_6 + x_7 = N \end{cases} \Rightarrow \begin{cases} x_6 + x_7 = N + 1 \\ x_6 + x_7 = N \end{cases}$$

which is a contradiction. Thus, $x_3 = x_{12} = 1$. Taking this into account and $x_8 = x_{10} \leq N(N-1)/2$ we can further reduce our set of constraints. Additionally, we can remove constants from the objective function. After simplification the ILP problem looks as

$$\begin{aligned} &\max_{x_5,x_6,x_7,x_8} (t_5x_5 + t_5x_6 + (t_7 + t_9)x_7 + (t_7 + t_{10})x_8) \\ &N + x_8 = x_5 + x_6 \\ &x_5 = x_7 + x_8 \\ &x_6 + x_7 = N \\ &x_8 \leq N(N-1)/2 \end{aligned} \tag{8}$$

and we know that $x_1 = x_3 = x_{12} = 1$, $x_2 = x_4 = x_{11} = N$, $x_9 = x_7$ and $x_{10} = x_8$.

Suppose that we have a solution $\{x_5, x_6, x_7, x_8\}$ of (8), for which $x_8 < N(N-1)/2$. One can easily see that this solution is not the optimal. The solution $\{x_5' = x_5 + 1, x_6' = x_6, x_7' = x_7, x_8' = x_8 + 1\}$ gives a larger value of the optimization function under assumption $t_5 + t_7 + t_{10} > 0$. For any realistic implementation the assumption is true. We can increment x_5 and x_8 until $x_8 = N(N-1)/2$. And this will give the maximum value of the optimization function for any particular x_6 and x_7. Thus, we conclude $x_8 = N(N-1)/2$.

Now the ILP problem looks as

$$\begin{aligned} &\max_{x_5,x_6,x_7} (t_5x_5 + t_5x_6 + (t_7 + t_9)x_7) \\ &N + N(N-1)/2 = x_5 + x_6 \\ &x_5 = x_7 + N(N-1)/2 \\ &x_6 + x_7 = N \end{aligned} \tag{9}$$

where the last constraint is redundant (it is a sum of two other constraints).

To solve problem (9) we express

$$\begin{aligned} x_6 &= N(N+1)/2 - x_5 \\ x_7 &= x_5 - N(N-1)/2 \end{aligned}$$

and put it to the optimization function:

$$\max_{x_5,x_6,x_7,x_8} (t_5x_5 + t_5x_6 + (t_7+t_9)x_7) =$$

$$\max_{x_5}\left(t_5x_5 + t_5\left(\frac{N(N+1)}{2} - x_5\right) + (t_7+t_9)\left(x_5 - \frac{N(N-1)}{2}\right)\right) \sim$$

$$\max_{x_5}(t_5x_5 - t_5x_5 + (t_7+t_9)x_5) =$$

$$\max_{x_5}((t_7+t_9)x_5) \sim$$

$$\max_{x_5} x_5$$

under assumption $t_7 + t_9 > 0$, which is true for any realistic implementation.

In order to maximize our optimization function we should maximize x_5, but this is infinity. At this point the feasibility constraint $x_6 \geq 0$ comes into a play:

$$x_6 = N(N+1)/2 - x_5 \geq 0 \quad \Rightarrow \quad x_5 \leq N(N+1)/2$$

Clearly, the maximal possible value of x_5 is $N(N+1)/2$. Finally, we find $x_6 = 0$, $x_7 = N$.

Our solution looks as following

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
1	N	1	N	$\frac{N(N+1)}{2}$	0	N	$\frac{N(N-1)}{2}$	N	$\frac{N(N-1)}{2}$	N	1

and the WCET is $\sum_{i=1}^{12} x_i t_i$. The solution does not depend on the exact values of $t_1, t_2, \ldots t_{12}$, only assumptions $t_5 + t_7 + t_{10} > 0$ and $t_7 + t_9 > 0$ are important. For $N = 32$ and cost values given in Fig. 1 the solution is

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	WCET
1	32	1	32	528	0	32	496	32	496	32	1	19571

The same values were obtaines with the `lp_solve` solver that was run on the originaly composed constraints (2–6).

2.3 Listings

2.3.1 `*.ais` files for arrays of different size

Array size 8

```
# enter AIS annotations here or use the AIS Wizard from the
    context menu
instruction "insertion_sort" is entered with @size = 8;
loop "insertion_sort" + 1 loop max @size begin;
loop "insertion_sort" + 2 loop max (@size - 1) begin;
label 0x1464 = "insertion_sort_move";

```

```
#flow at most @size x (@size -1)
flow "insertion_sort_move" <= 56 ("insertion_sort");#size 8
```

Array size 16

```
# enter AIS annotations here or use the AIS Wizard from the
    context menu
instruction "insertion_sort" is entered with @size = 16;
loop "insertion_sort" + 1 loop max @size begin;
loop "insertion_sort" + 2 loop max (@size - 1) begin;
label 0x1464 = "insertion_sort_move";

#flow at most @size x (@size -1)
flow "insertion_sort_move" <= 240 ("insertion_sort");#size 16
```

Array size 32

```
# enter AIS annotations here or use the AIS Wizard from the
    context menu
instruction "insertion_sort" is entered with @size = 32;
loop "insertion_sort" + 1 loop max @size begin;
loop "insertion_sort" + 2 loop max (@size - 1) begin;
label 0x1464 = "insertion_sort_move";

#flow at most @size x (@size -1)
flow "insertion_sort_move" <= 496 ("insertion_sort");#size 32
```

Array size 64

```
# enter AIS annotations here or use the AIS Wizard from the
    context menu
instruction "insertion_sort" is entered with @size = 64;
loop "insertion_sort" + 1 loop max @size begin;
loop "insertion_sort" + 2 loop max (@size - 1) begin;
label 0x1464 = "insertion_sort_move";

#flow at most @size x (@size -1)
flow "insertion_sort_move" <= 4032 ("insertion_sort");#size 64
```

2.3.2

No listing for this question

2.3.3 -O1

```
# enter AIS annotations here or use the AIS Wizard from the
    context menu
instruction "insertion_sort" is entered with @size = 32;
loop "insertion_sort" + 1 loop max @size begin;
loop "insertion_sort" + 2 loop max (@size - 1) begin;
label 0x12fc = "insertion_sort_move";

#flow at most @size x (@size -1)
flow "insertion_sort_move" <= 496 ("insertion_sort");#size 32
```

2.3.4 Input file for ILP solver

```
/* Run with "lp_solve p2q4.lp" */

/* Constants:
  t1  = 12
  t2 = t3 =  6
  t4  = 15
  t5 = t6 =  4
  t7 = t8 = 11
  t9  =  1
  t10 = 21
  t11 = 16
  t12 =  1
  N = 32
*/

/* Objective: maximize WCET */
max: WCET;


/****** Constraints begin *******/
/* WCET definition: WCET = SUM{i=1..12}(t_i*x_i) */
WCET = 12 x1 + 6 x2 + 6 x3 + 15 x4 + 4 x5 + 4 x6 + 11 x7 + 11 x8 +
     1 x9 + 21 x10 + 16 x11 + 1 x12;
/* Exactly one execution */
x1 = 1;
/* Vertices constraints */
x2 + x3 = x1 + x11;
x2 = x4;
x4 + x10 = x5 + x6;
x5 = x7 + x8;
x7 = x9;
x8 = x10;
x6 + x9 = x11;
x3 = x12;
/* Termination or infinite execution */
x12 <= 1;
/* Number of outer loop iterations: x11 = N */
x11 = 32;
/* Maximum number of inner loop iterations x10 <= N*(N-1)/2 */
x10 <= 496;
/* Physical feasibility: xi >= 0 --> assumed implicitly with
    variable type 'int' */
/******* Constraints end ********/
```

```c


/* Declarations: we are working with integers */
int WCET, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12;
```

3 Problem 3

3.1 Problem Statement

First, focus on the analysis of `task.c:merge_samples()`. The loop bounds in this function depend on the input-data dependent parameter `@inputcount`, defined to be the maximal value of `input->input_count`. Add loop bounds and flow facts for `merge_samples`, and analyze its WCET assuming that `@inputcount` ≤ 64.

Q1: Describe and justify the loop bounds and flow facts used in the WCET analysis of `merge_samples`.

Q2: What is the maximum execution time observed for this function? What is the analyzed WCET? In case they do not coincide, discuss reasons for the impreciseness in the static analysis.

Hint: For measurements, you might want to rely on or start with the test bench implemented in
`main.c:test_merge_samples()`.

Hint: Division is implemented in software for the Patmos architecture. We replaced the division in the interpolation routine by a lookup table for small values, to simplify the analysis problem. You do not need to worry about the analysis of `iinterpolate16`; simply use the following annotations:

```
include "llvm.ais";
instruction "iinterpolate16" + 1 calls is never executed;
instruction "iinterpolate16" + 2 calls is never executed;
```

3.2 Solution

A1: To justify flow and loop constraints for `merge_samples` we first need to state the following facts: number of samples that are used for analysis is 64 (`SAMPLE_COUNT`). At most 4 consecutive samples could be missing (`MAX_CONSECUTIVE_MISSING`). This means that minimum number of valid samples to perform the analysis is:

$$\text{MIN}_{\text{valid samples}} = \lceil \frac{\texttt{SAMPLE\_COUNT}}{\texttt{MAX\_CONSECUTIVE\_MISSING} + 1} \rceil$$

(in our case 13).

`merge_samples` has two nested loops and two conditional branches of interest (we do not consider the case when number of samples <= 0, where we exit without performing interpolation).

The outer loop is executed for each sample (either valid or missing, 64 times in total, line 3 of `ait_merge_samples` in the listing). Then we hit the first `if` statement: take the branch if current value is not missing. All values might be valid, hence we can take the branch 64 times (`ait_merge_samples`: lines 6 and 11). Now, the current sample is valid (line TODO in `merge_samples`, see listing), but several previous (up to 4) samples might be missing. We take the branch in the second `if` condition (`merge_samples`: line TODO) if current sample is valid and one or several previous samples are missing (the worst case when each second sample is missing – we can take the branch max 32 times – `ait_merge_samples`: lines 7 and 12). We will use this fact as a flow constraint. The inner loop interpolates missing samples: at most 4 might be missed consecutively, hence the loop bound for the inner loop is 4 (`ait_merge_samples`: line 4). Max number of iteration of the inner loop:

$$\text{MIN}_{\text{valid samples}} \times \text{MAX\_CONSECUTIVE\_MISSING}$$

which is equal to 52 (`ait_merge_samples`: lines 5 and 10).

A2: Measurements for test bench implemented in `main.c:test_merge_samples()` are shown in the Figure 4. The result of static analysis using `a3patmos` gives the following worst case execution bound:

$$\text{WCET}_{\text{merge\_samples}} = 2\ 024\ 556$$

3.3 Listings

3.3.1 ait_merge_samples.ais

```
1  # enter AIS annotations here or use the AIS Wizard from the
       context menu
2  instruction "merge_samples" is entered with @inputcount = 64;
3  loop "merge_samples" + 1 loop max @inputcount begin;
4  loop "merge_samples" + 2 loop max 4 begin;
5  label 0x2878 = "merge_samples_interpolate";
6  label 0x2798 = "merge_samples_valid_values";
7  label 0x2800 = "merge_samples_buffer_get";
8
9
10 flow  "merge_samples_interpolate" <= 52 ;
11 flow  "merge_samples_valid_values" <= 64;
12 flow "merge_samples_buffer_get" <= 32;
```

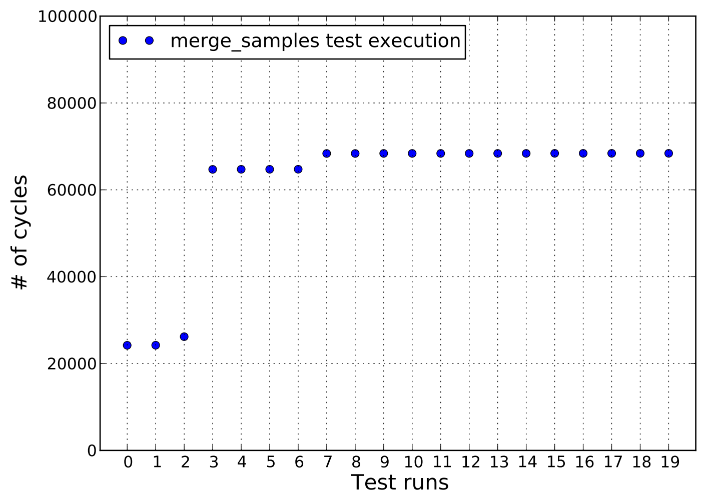


Figure 4: Measurements for `merge_samples()`

3.3.2 task.c:merge_samples()

```c
/*
    ***************************************************************************

 * Add (at most) in->inputcount new samples to the sample buffer
 * Interpolate missing samples if possible

    **************************************************************************
    */
void merge_samples(input_t* in, sample_buffer_t* sbuf)
{
  int i, j, cnt, valid;
  sample_value_t  x;
  sample_value_t *xs;

  /* return if input has no samples */
  if(in->input_count <= 0) return;

  cnt = in->input_count;
  xs = in->input_samples;
  valid = sample_buffer_get_valid(sbuf);

  for(i = 0; i < cnt; i++)
  {
    /* ai: loop here max inf; */
    x = xs[i];

    sample_buffer_set(sbuf,i,x);
    /* If the sample is not missing, interpolate the ones before
   if the range is acceptable */
    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)
      {
        /* Calculate interpolated value for all samples in the
   range [valid+1,i-1] */
        int16_t z = sample_buffer_get(sbuf,valid);
        for(j = i-1; j > valid; --j)
        {
    /* ai: loop here max inf; */
          /* ai: label here = "merge_samples_interpolate"; */
          int16_t y = iinterpolate16(valid,z,i,x,j);
          sample_buffer_set(sbuf,j, y);
        }
      }
      valid = i;
    }
  }
  /* increment ring buffer index */
  sample_buffer_set_valid(sbuf, valid);
  sample_buffer_incr_ptr(sbuf, cnt);
  return;
}
```

4 Problem 4

4.1 Problem Statement

Next, 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, and add a flow constraint relating the execution frequency of the inner loop's body with the function's execution frequency.

Q1: Describe and justify the flow facts found for `fp_radix2fft_withscaling`, and `fft`.

Q2: What is the maximum execution time observed for this function? What is the analyzed WCET?

Q3: For the FFT implementation `fp_radix2fft_withscaling`, is it safe to use the loop iteration counts observed during a test run? Justify your answer.

Hint: For measurements, you might want to rely on or start with the test implementation `main.c:test_fft()`

4.2 Solution

A1: Function `fp_radix2fft_withscaling` represents a classical implementation of Fast Fourier Transform algorithm. This algorithm can be implemented in hardware. This claim can be easily checked by inspecting the code. A feature of these kind of algorithms is that the function code can be fully unfolded in a plain linear program, i.e. control flow does not depend on the input data. We treat function arguments `n` and `t` as parameters of the algorithm. In our problem they are constant: $n = 64$, $t = \log_2 n = 6$.

For every loop in `fp_radix2fft_withscaling` it is true that the loop variable does not change inside the loop body. The outer loop executes exactly $t = 6$ times. The two inner loops execute at most $2^5 = 32$ times each during a single iteration of the outer loop. Moreover, the bounds of the two inner loops are set exclusively in the outer loop and in the way that there is a strict relation between them. Indeed, if the current iteration of the outer loop is q, then the number of iterations of the inner loops are exactly $2^{t-q} = 2^{6-q}$ and 2^{q-1}. The number of the most inner loop body executions is exactly $2^{6-q} \cdot 2^{q-1} = 2^5$ in the current iteration of the outer loop. In total, the most inner loop body is executed exactly $t \cdot 2^{t-1} = 6 \cdot 2^5 = 192$ times. The middle loop (the inner loop with the loop variable `k`) body execution count depends on the value of `r` which is constant during each outer loop iteration, but decreases after each outer loop iteration. In total, the middle loop body is executed exactly $\sum_{q=1}^{t} 2^{t-q} = 2^t - 1 = 2^6 - 1 = 63$ times.

Function `fp_radix2fft_withscaling` performs two calls of the function `bitreverse` before entering the loops. However, `bitreverse` has the same properties as `fp_radix2fft_withscaling` itself: the control flow does not depend on the data, the loops can be fully unfolded and the function can be implemented in hardware. We treat a constant array, passed to the function, as a parameter, but not an input, since it is always the same for all function calls with the sane value of `n`. In fact, `bitreverse` represents constant permutation or fixed routing in hardware. Clearly, the loop is always executed $n = 64$ times, but the body of the `if` operator is expected to be executed significantly smaller number of times. Although, it is difficult to compute this number analytically, the function will always follow the same control flow. With a single run of the function we found that the body of the `if` operator is executed 28 times.

In order to define loop bounds for `fft` we exploit the following observation: the function has four loops, which are not nested. First loop executed at most 4 (?5 TODO) times (`ait_fft`: line 2, see listing below). Loops 2, 3 and 4 are executed 64 (or `SAMPLE_COUNT`) times each (`ait_fft`: lines 3-5).

For `fp_radix2fft_withscaling` we use loop bounds and flow facts discussed above (`ait_fft`: lines 7 - 23).

A2: Since `test_fft()` provides only one execution of `fft` we modified `merge_samples()` to perform more elaborate testing. We invoke `fft` 20 times with different `input_buffer`. The measurements for `fp_radix2fft_withscaling` and `fft` are shown in the Figure 5.

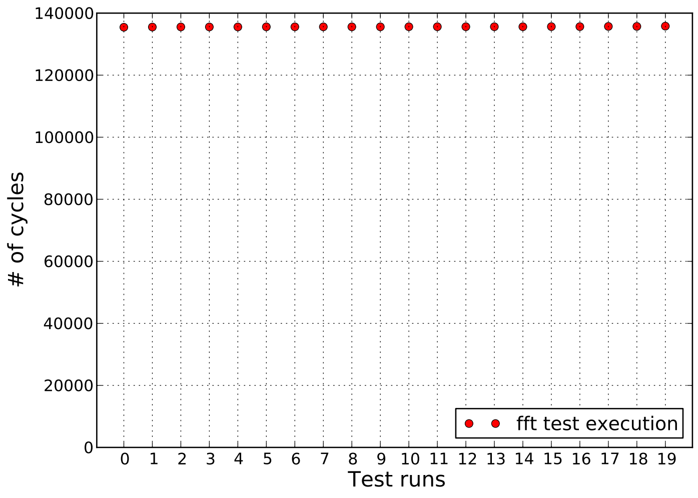
WCET analysis with `a3patmos` gives the following results:

$$\text{WCET}_{\text{fp\_radix2fft\_withscaling}} = 3\ \ 496\ \ 504$$
$$\text{WCET}_{\text{fft}} = 4\ \ 919\ \ 107$$

A3: It is absolutely safe to use for analysis the loop iteration counts observed during a test run of `fp_radix2fft_withscaling`. The reason is given in the section A1 above.

4.3 Listings

4.3.1 ait_fft.ais



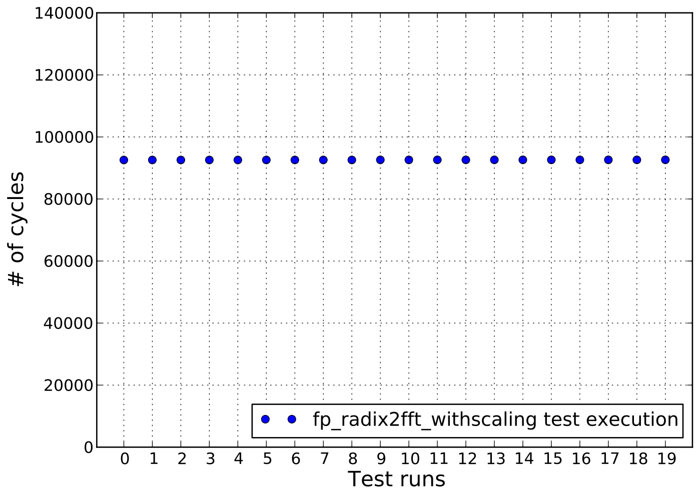


Figure 5: Measurements of execution time

```
instruction "fft" is entered with @samplecount = 64;
loop "fft" + 1 loop max 5 begin;
loop "fft" + 2 loop max @samplecount begin;
loop "fft" + 3 loop max @samplecount begin;
loop "fft" + 4 loop max @samplecount begin;

instruction "fp_radix2fft_withscaling" is entered with @n = 64;
instruction "fp_radix2fft_withscaling" is entered with @t = 6;
loop "fp_radix2fft_withscaling" + 1 loop max @t begin;
loop "fp_radix2fft_withscaling" + 2 loop max 32 begin;
loop "fp_radix2fft_withscaling" + 3 loop max 32 begin;
```

```

#second loop in fp_radix2fft_withscaling -- flow at most 63 times
label 0x1be8 = "radix2fft_second_loop";
flow "radix2fft_second_loop" <= 63 ("fp_radix2fft_withscaling");

#third loop in fp_radix2fft_withscaling -- flow at most 192 times
label 0x19a4 = "fp_radix2fft_withscaling_body";
flow "fp_radix2fft_withscaling_body" <= 192 ("
    fp_radix2fft_withscaling");


instruction "bitreverse" is entered with @n = 64;
loop "bitreverse" + 1 loop max @n begin;
```

4.3.2 fft

```
/*
    ****************************************************************************

 * FFT transform, N = SAMPLE_COUNT

    ****************************************************************************
    */
status_t fft(sample_buffer_t *buf, int16_t * fft_r_out, int16_t *
    fft_i_out)
{
  int i, offs;
  sample_value_t max = 0;
  int32_t multiplier;

  for(offs = -1; offs >= -MAX_CONSECUTIVE_MISSING; --offs)
  {
    /* ai: loop here max 5; */
    if(! IS_VALUE_MISSING(sample_buffer_get(buf,offs))) break;
  }
  if(offs < -MAX_CONSECUTIVE_MISSING) return
    S_TOO_FEW_VALID_SAMPLES;

  /* Copy and normalize real-valued input */
  offs -= SAMPLE_COUNT-1;

  for(i = 0; i < SAMPLE_COUNT; i++)
  {
    /* ai: loop here max @samplecount; */
    int16_t val = sample_buffer_get(buf,offs+i);
    if(IS_VALUE_MISSING(val)) return S_TOO_FEW_VALID_SAMPLES;
    if(val < 0) val = -val;
    else if(val > max) max = val;
  }
  if(max != 0)
  {
    multiplier = (((int32_t)NORM_MAX)<<16) / max;


    for(i = 0; i < SAMPLE_COUNT; i++)
    {
      /* ai: loop here max @samplecount; */
      /* Normalize real-valued input */
      int32_t old_val = sample_buffer_get(buf,i-SAMPLE_COUNT);
      fft_r_out[i] = (int16_t)((old_val*multiplier) >> 16);
    }
  }

  /* Set imaginary input to 0 */
  /* memset(&fft_i_out[0], 0, SAMPLE_COUNT * sizeof(fft_i_out[0]))
   ; */


  for(i = 0; i < SAMPLE_COUNT; i++)
  {
    /* ai: loop here max @samplecount; */
    fft_i_out[i] = 0;
  }

  /* Run fixed-point FFT */
  /* ai: instruction here is entered with @n = 64; */
  /* ai: instruction here is entered with @t =  6; */
  fp_radix2fft_withscaling(
    &fft_r_out[0], &fft_i_out[0],
    BIT_REVERSE64, TWIDDLE_R64, TWIDDLE_I64,
    SAMPLE_COUNT, SAMPLE_COUNT_LOG2);
  return S_OK;
}
```

4.3.3 fp_radix2fft_withscaling

```
/** @brief fixed point FFT
  * @param xr  real part of length n (input/output)
  * @param xi  imaginary part of length n (input/output)
  * @param brv bit reverse matrix of length n
  * @param wr  real part of twiddle factors of length n
  * @param wi  imaginary part of twiddle factors of length n
  * @param n   length of the input (user register @n)
  * @param t   log2(n) (user register @t)
  *
  * adapted from: http://www.mathworks.com/products/fixed/demos.
   html?file=/products/demos/shipping/fixedpoint/fi_radix2fft_demo
   .html
  *
  */
void fp_radix2fft_withscaling(int16_t* xr, int16_t* xi,
                              const int *bvr,
                                    const int16_t* wr, const int16_t* wi
    ,
                                    int n, // @n = 64
                                    int t) // @t = 6
{
  int32_t tempr, tempi;
  int q, j, k;
  int n1, n2, n3;
  int L, kL, r, L2;

  bitreverse(xr,bvr,n);
  bitreverse(xi,bvr,n);

  /*  Work out the loop bounds for the FFT transform */


  for (q=1; q<=t; q++)
  { /* ai: loop here max 6; */
    L = 1; L <<= q;
    r = 1; r <<= (t-q);
    L2 = L>>1;
    kL = 0;


    for (k=0; k<r; k++)
    { /* ai: loop here max 32; */
    /* ai: flow here = 63; */
      for (j=0; j<L2; j++)
      {
    /* ai: loop here max 32; */
      /* ai: flow here = 192; */
  n3      = kL + j;
  n2      = n3 + L2;
  n1      = L2 - 1 + j;
  /* ai: label here = "fp_radix2fft_withscaling_body"; */
  tempr   = (int32_t)wr[n1]*(int32_t)xr[n2] - (int32_t)wi[n1]*(
    int32_t)xi[n2];
  tempi   = (int32_t)wr[n1]*(int32_t)xi[n2] + (int32_t)wi[n1]*(
    int32_t)xr[n2];
  xr[n2]  = ((((int32_t)xr[n3])<<FP_FRAC) - tempr)>>(FP_FRAC+1);
  xi[n2]  = ((((int32_t)xi[n3])<<FP_FRAC) - tempi)>>(FP_FRAC+1);
  xr[n3]  = ((((int32_t)xr[n3])<<FP_FRAC) + tempr)>>(FP_FRAC+1);
  xi[n3]  = ((((int32_t)xi[n3])<<FP_FRAC) + tempi)>>(FP_FRAC+1);
      }
      kL += L;
    }
  }
}
```

4.3.4 bitreverse

```c
/** @brief Permutation of the elements in vs for FFT
 *  @param vs the value vector
 *  @param brv the bitreverse lookup table
 *  @param n the size of vs and brv
 *  User-Register: @n >= n
 */
static void bitreverse(int16_t* vs, const int* brv, int n)
{
  int i;
  for(i = 0; i < n; i++)
  {
    /* ai: loop here MAX (@n); */
    int j = brv[i];
    if(i < j)
    {
      int16_t tmp = vs[i];
      vs[i]       = vs[j];
      vs[j]       = tmp;
    }
  }
}
```

5 Problem 5

5.1 Problem Statement

The final goal is to analyze the WCET of `task.c:task()`.

Q1: First, calculate the WCET of `check_sin` and `check_square`. Then, try to give a rough estimate on the WCET by combining the WCET's of `merge_samples`, `fft`, `check_sin` and `check_square`. Clearly state how you combined the WCETs, and compare the resulting value with the maximum observed execution time of these functions, combined in the same way.

Q2: Add a flow fact for the indirect call in `task`, calculate the WCET of `task`, and compare it to the maximum execution time observed.

5.2 Solution

Function	WCET (cycles)
merge_samples	2152787
fft	4905711
check_sin	2318892
check_square	2386340

Table 1: The individual WCETs of the observed functions.

A1: In Table 1 we observe WCETs of all functions including *check_sin* and *check_square*. The flow path of the function *task* consist out of one call the functions merge_samples and fft each, and either one call of check_sin or check_square. In on or estimation of $WCET_{task}$ we chose a combination of $WCET_{merge\_samples}$,$WCET_{fft}$ and $WCET_{check\_square}$ because their sum yields the largest execution time.

$$\begin{aligned} WCET_{Task} &= WCET_{merge\_samples} + WCET_{fft} + WCET_{check\_square} \\ &= 9444878 \quad \textit{cycles} \end{aligned} \tag{10}$$

WCET	cycles
$WCET_{estimated}$	9444878
$WCET_{meassured}$	9453086

Table 2: Estimated and measured $WCET_{task}$

A2: Table 2 shows comparison between estimated and measured WCET of the function *task*.

5.3 Listings

6 Reflections

Answer the following questions:

Q1: How much time did you spend writing annotations and analyzing the code? Was it less or more than you expected?

Q2: What problems did you encounter during this assignment?

Q3: As you learned, sometimes it is necessary to annotate the assembly code. Why? What problems can you see because of this?

6.1 Answers

A1: We have spent quite some time doing the lab, and (at least for me) it was more than expected. Writing annotations will not take much time for a person familiar with the framework and the work flow, but familiarising yourself with tools takes time. In addition, documentation is spread across different files and places. The claim in the lab assignment that “source code annotation sometimes might not work” was the reason for us to use separate `*.ais` files instead.

A2: Problems: optimizations, performed by compiler affect the structure of CFG in `a3patmos`: it gave wrong instruction address, to resolve the correct result analysis of the disassembly needed.

A3: The analysis is architecture-specific, and assembly code is exactly what is executed by the hardware. It is sometimes necessary annotate assembly code since code structure might be altered during compilation. Annotations of assembly instructions complicate the analysis (and may become a source of additional errors) since one needs to map the knowledge about the program to assembly code. In this case we see the following problems: analysis takes more time, and it becomes more error-prone.