WCET Analysis Lab: Assignment 1

Due on 18.4.2011

v1.0

Benedikt Huber, Peter Puschner

Introduction

In the first assignment, you are going to analyze the execution time of a few functions on a simple architecture, and think about the challenges when analyzing more complex real-time tasks. We will use a very simple LEON3 configuration (no caches) in the first assignment, and focus on high-level WCET analysis (path analysis). As a general recommendation, for each problem, you should do the following:

(i) **General Analysis:**
Before analyzing the execution time, you should carefully examine what kind of input data influences the tasks behavior. Think about the set of possible execution trace, and the influence of input data on control flow decisions.
You should identify all input variables and all compile time constants which are necessary to compute the execution time of the task, or might be useful to do so.

(ii) **End-to-end measurements:**
Measuring the execution time of a task or function (from its beginning to its end) should give you an intuition about the expected execution times (depending on the input data). These end-to-end measurements rely on test cases with a good execution path coverage, though it is usually intractable to cover all possible execution paths.
You should design a set of testcases for end-to-end measurements and then measure execution times on the DE2-70 board.

(iii) **Static WCET Analysis:**
Static WCET Analysis needs a precise model of the set of possible execution paths. In addition to clever automated analyses, with todays tools manual annotations are often necessary to reconstruct the control flow graph, bound the number of loop iterations, or exclude other infeasible paths to obtain more precise results.
You should add annotations describing the control flow to the source. Then, calculate safe upper bounds to the WCET using the analysis tool *aiT*. To this end, it might be necessary to add annotations referring to assembler code.

(iv) **Report:**
You should report on the problem, the measurement and static analysis results, and answer the problem question using the measurement and analysis results you've obtained.

Resources

- The sources of the code you are going to build and analyze is available from the course homepage [1]
- The introduction to the lab environment for the first exercise [2]
- The flow fact annotation primer [3]
- The aiT manual [4]

Deliverables

- A report with short answers to the problems, measurement and analysis results, and a summary of what has been done in the source code.

[1] `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/Laborteil`
[2] `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/ressourcen/timing-analysis-lab-first-steps`
[3] `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/ressourcen/ait-flow-fact-annotation-primer`
[4] Available in the Lab and via myTI

- The sources of the functions performing end-to-end measurements.
- The `.ais` files used in the static analysis, and the updated sources with flow annotations.

Points and Grading In total, you can achieve 35 (regular) points in the first lab, plus up to 10 bonus points.

Getting Started (6)

Problem 1

Recommended (3): As a warm-up exercise, follow the instruction in "Timing Analysis Lab: First Steps"[5].
Q: How long does it take to execute `simple` once, according to measurements, and according to the static analysis?

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?

Introduction to WCET Analysis (9)

Here you are going to analyze the WCET of the function `insertion_sort` [6]. You should write tests and measure the execution time of `insertion_sort.c:insertion_sort()`, and extract flow facts in order to analyze the WCET statically.

Simple Measurements Due to the potentially huge number of possible execution paths, and due to data and history dependent hardware timing, simply measuring the WCET of the task is usually intractable. However, it is important to perform measurements to relate the computed WCET with the largest observed execution time. For parts of the task, it should be possible to get very close to the analyzed WCET, which is a *test* for both the quality and correctness of flow facts and the WCET analysis tool itself.

The easiest way to perform end-to-end measurements is to use the macros and functions defined in the file `bench.h`:

- The function `init_benchmark(int verbose)` sets up the timers, and computes an estimate for the expected measurement overhead.
- The macro `MEASUREMENT_START()` starts the measurement, storing the initial timer value in the global variable `start` (attention: each timer tick corresponds to TIMER_GRANULARITY (4) cycles).
- The macro `MEASUREMENT_STOP()` stops the measurement, storing the timer value at the end of the measurement in the global variable `stop`.
- The macro `ELAPSED_CYCLES()`, when called after a measurement, returns the number of elapsed cycles.
- The macro `MEASURE_OVERHEAD()` returns an estimation for the measurement overhead.

Note: Do not forget to call `init_benchmarks` at the beginning of your test program

[5] `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/ressourcen/timing-analysis-lab-first-steps`
[6] In the `insertion_sort` folder of the sources for the first assignment

Getting Started with static WCET analysis Please have a look at the flow fact annotation primer. [7]. Further questions are answered in the aiT manual available in the lab.

Note: Do not forget that you (a) need to enable extraction of source code annotations in aiT, and (b) that source code annotations sometimes might not work due to imprecise debugging information. In the latter case, inspect the control flow graph of the function, add annotations in the `.ais` *files, using machine code addresses instead of symbolic position markers*

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?

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? Q: What results do you get for an array size of 8,16 or 64, using measurements and static analysis? Q: In addition to the size of the array, what other aspects of the input data might influence the WCET?

Analyzing the signal monitor benchmark (16)

The "signal_monitor" benchmark analyzes the output of signal generators and reports whether they perform within their specified range, or if they are broken (`task.c:task()`) [8]

The check is executed for each signal, and proceeds as follows:

- The samples are collected asynchronously, so as a first step, all new samples have to be stored in the internal buffer. The buffer has a capacity of 128 samples, but only the 64 most recently added are used for the analysis. When merging, missing samples are guessed using linear interpolation (`task.c:merge_samples()`).
- The task then performs a fixed-point Fast Fourier Transform, first normalizing the sample values. This results in (a part of) the frequency spectrum of the sampled signal (`task.c:fft()`).
- Finally, for each signal a check is run to guess whether the generator is faulty or not. The kind of check which is executed varies from signal to signal and is passed as parameter `signal_spec` to `task.c:task()`.
  - For sinus signals, the task checks that the average value is below a certain threshold, and whether there is one and only one frequency with a high amplitude. (`task.c:check_sin()`).
  - For square wave signals, the (simple) check verifies that the high amplitude frequencies are in decreasing order, and above a minimum threshold (`task.c:check_square()`).

[7] `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/ressourcen/ait-flow-fact-annotation-primer`
[8] The benchmark is found in the monitor_signal directory

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.
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).*

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 function's 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. 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. Does the FFT loop bound depend on the input data?

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.

Reflections (Mandatory)

Problem 8

Mandatory (4): Answer the following questions:

- 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?
- What is the ratio between observed and actual execution time? Discuss the causes of the overestimation.
- As you learned, sometimes it is necessary to annotate the assembler code. Why? What problems can you see because of this?

.