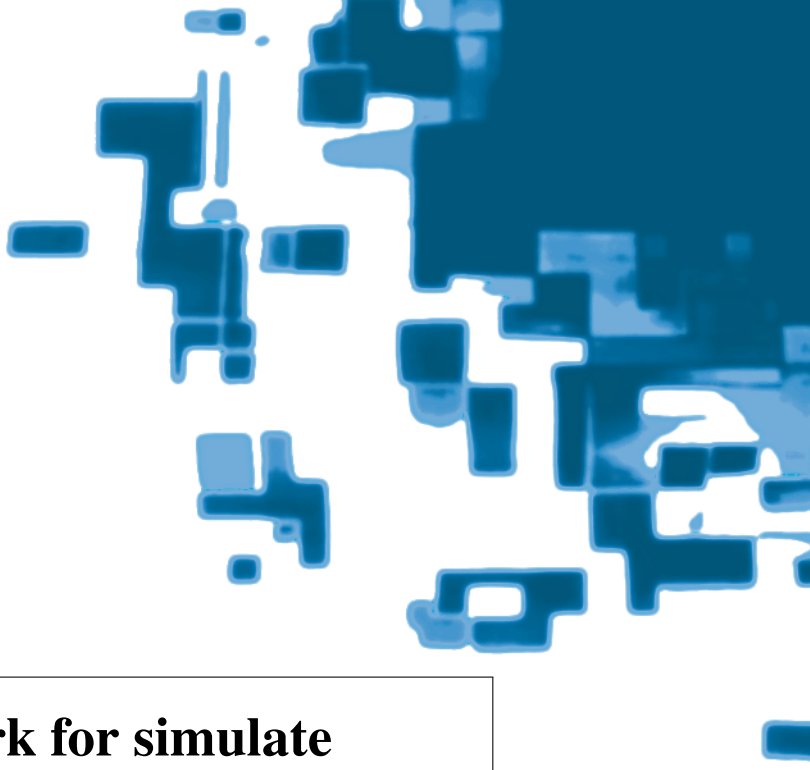

A framework for simulate synchronous reactive programs and measure execution times to aid WCET analysis

Wei-Tsun Sun

Verimag Research Report n^o TR-2016-3

2016-07-20

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UGA
Bâtiment IMAG
Université Grenoble Alpes
700, avenue centrale
38401 Saint Martin d'Hères
France
tel : +33 4 57 42 22 42
fax : +33 4 57 42 22 22
http://www-verimag.imag.fr/

A framework for simulate synchronous reactive programs and measure execution times to aid WCET analysis

Wei-Tsun Sun

2016-07-20

Abstract

Obtaining Worst Case Execution Time (WCET) is essential for systems with timing requirement. This is because the violation of such requirement may lead to catastrophic results in safety critical systems. WCET can be acquired through static or dynamic analyses. With static analysis, it is less obvious to observe the timing behaviors of the analyzed system during its actual execution. While dynamic analysis, such as measuring execution times, lacks of theoretical foundation to ensure the specified properties. It is essential to perform the static and dynamic analyses for the same hardware configuration to have consistent results and meaningful feedback. It is equally essential to have both analysis perform on the same analysis framework for the same reason. In this paper, we are presenting an approach of dynamic analysis through the use of a framework, named OSIM, which integrates the processor simulator and input-stimuli generator. OSIM is based on OTAWA, a static timing analysis framework, and is able to give feedback to OTAWA for the refinements of the static analysis.

Keywords: WCET, measurement WCET analysis, static timing analysis

Reviewers: Claire Maiza, Pascal Raymond, Erwan Jahier

How to cite this report:

```
@techreport {TR-2016-3,
   title = {A framework for simulate synchronous reactive programs and measure execution
times to aid WCET analysis},
  author = {Wei-Tsun Sun},
  institution = {{Verimag} Research Report},
  number = {TR-2016-3},
  year = {2016}
}
```

1 Introduction

In real-time systems, in particular the hard real-time systems, the programs have to react to the environment in a given time (as a constraint) to prevent possible failure of the system. To ensure the execution time of the program never excesses such constraints, a worst case execution time (WCET) analysis is performed. WCET analysis can be categorized into two major groups [21]: the static analysis and the dynamic analysis (also called measurement approach). The static analysis estimate a guaranteed upper-bound on the execution time of a given program. Over-estimation of the bound can occur, for instance when the analysis does not exclude the infeasible paths, i.e. the execution states which are impossible to reach. The statically WCET analysis does not provide any information about the execution time depending on different execution scenarios. The estimated bound is provided for any input value (worst-case). Execution times of a program are measured in the dynamic analysis. The measurements are taken from numerous execution scenarios with different inputs fed to the programs. To achieve higher confidence in the dynamic analysis, the number of the execution scenarios should be sufficient to provide well-representative samples, while the inputs to the program also need to be well selected to cover as many execution scenarios as possible. Dynamic analysis provides the possible distribution of the execution times, but the upper-bound found in the measurements may be exceeded in the case where the measurement does not cover all scenarios. Static and dynamic analysis can be applied to aid each other (each one giving feedback to the other one).

In this report, we focus on synchronous programs, in particular the programs that are written in a synchronous language [1] called Lustre [8]. Synchronous languages like Lustre describe reactive systems which continuously react to the environment. The features of Lustre achieve the determinism in both functional and temporal manner, which is suitable for targeting the real-time systems. A Lustre program is compiled to the binary of the target platform. This binary is then used by our analysis. For the dynamic analysis, in order to obtain confident measurements, the program is executed/simulated intensively so that the results can reflect the actual behavior of the system in the real environment. Sensible inputs also needs to be provided to the simulated program to have meaningful program behaviors and outcomes. This can be achieved with the help from Lurette [13], which provides the infrastructure to integrate both the stimuli generator and the simulated programs (we can them system under test, or SUT) running in OSIM. In Lurette, Lutin [12, 11, 16] is used to implement the stimuli generator which feeds randomized inputs to the simulated programs by specifying the possible input values.

In this report, we present a simulator OSIM to be used to give feedback to static analysis. This simulator is connected to an environment generator through Lutin/Lurette that enables a good dynamic analysis of Lustre programs.

There exists several tools for performing static analysis. We use OTAWA [4], which enables us to perform static analysis for the ARM7 platform. In OTAWA, the target hardware is configured prior to the analysis. To have a clear picture of the execution times of the programs, we decide to apply dynamic analysis as the counterpart to the results obtained by OTAWA. The factors that affect the execution time depends on the hardware such as the characteristic of the processor, the memory hierarchy, the infrastructure of the bus, and the peripherals (i.e. inputs and outputs). Hence we use the same hardware configuration and mechanism provided by OTAWA to avoid the influences from having the different hardware platform. We construct a simulator, named OSIM, which executes the program binaries on the simulated hardware platform to measure the execution times.The results collected from the dynamic analysis are then compared with the ones from static analysis as the feedback to investigate the accuracy of the static analysis and the possible improvements. With the stimuli generator (Lurette/Lutin) we show that a good environment model helps a lot in getting correct and more precise dynamic WCET. The report is organised as follows: TODO

2 Related works

This work is clearly related to the abundant literature on test input generation, for instance: random test generation [20], genetic algorithm [17], model-checking [20], path clustering [5] or profiling [9]. The main difference in our approach is that we focus more on the feasibility and relevance of inputs, where other approaches generally focus on code coverage.

Performing dynamic WCET measurement with the help of a model of the environment does not seem

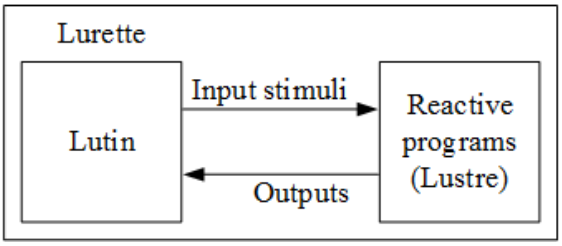

Figure 1: Testing Lustre reactive programs with Lutin in Lurette

common. To our knowledge, the most similar approach is presented in [6]. The main difference with our work is that the environment is described (and simulated) with Matlab-Simulink. Simulink is well suited for modeling continuous time, deterministic, physical environment. Lutin which is specifically designed for testing purpose, is more suitable and versatile for describing and simulating sequential, non deterministic scenarios: with a compact description and a intensive automatic testing, the Lutin framework can discover rare/borderline executions.

A main goal of this work is to quantify the precision of static WCET estimation. The chosen method is the comparison with dynamic WCET measurement performed - in the same abstract machine, for the same set of realistic environment. A similar approach exists in the tool Chronos [15], but but the constraints on inputs are much simpler than the one we can describe with Lutin. Other methods aim at quantifying the precision of estimation using specific analysis [3].

3 Background and our contributions

3.1 Timing analysis of programs

The execution times of programs depend on the inputs of the program. This is because different inputs scenarios will form different conditions which results different execution paths. The behaviors of the programs can also depend on the current state of the program, i.e. mealy machine like. It is then very difficult to explorer manually all the combinations of the inputs and the program states to find the worst-case execution-time (WCET) of the program. Static WCET analysis does not take different input sets into account and estimate the WCET by finding the corresponding execution path. Static WCET analysis does not provide the distributions of the execution times (neither the best nor the nominal cases) hence it is difficult to relate the WCET with the behaviors of the programs.

3.2 Testing reactive programs

Synchronous reactive languages such as Lustre are used to design real-time systems. For critical systems, both functional and timing correctness are equally important. Lurette [13] is framework which enables designer to automatically test Lustre programs for functional correctness. The infrastructure of Lurette is depicted by Figure 1. Designers can design input generators in Lutin, to provide input stimuli to the system under test (SUT), i.e. the reactive programs. With Lutin, the designers does not need to come up with individual input scenarios, instead, the inputs will be generated automatically by specifying the constraints of the inputs. For example, a program may take two Boolean values as inputs but it is impossible to have both values equal to true. It is then possible to apply this restriction when designing the input stimuli with Lutin to generates all possible input cases (true/false, false/true, and false/false). With proper techniques, the designers can described the rules to generate the inputs with Lutin, and let Lutin provides the actual inputs to the SUT. The SUT will generate outputs based on the given inputs, where the outputs can be fed back to Lutin for verifying the functional correctness. However, checking the timing properties of a synchronous program is not available in Lurette.

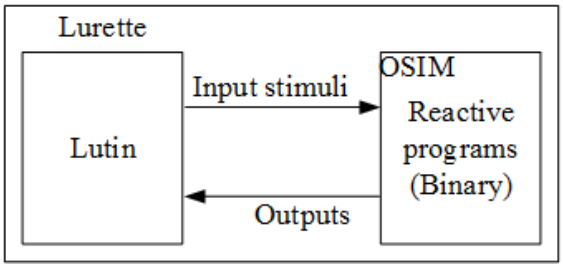

Figure 2: Simulating the executable reactive programs with OSIM in Lurette

3.3 Contributions

In this report, we propose a framework to perform measurement approach to obtain WCET of the Lustre programs. For start, we are interested particularly in the ARM7 LPC2138 architecture, however the proposed method is not architecture dependent. We use Lutin to generate vast input sets automatically to obtain a wide range of measurements. Instead of running the compiled Lustre programs on the real hardware platform, the programs are executed on a simulator named OSIM (short for OTAWA SIMulator) as illustrated in Figure 2. OSIM is developed based on a framework called OTAWA which provides many features mainly designed for static analysis. In this case, the same facilities, such as control flow graph (CFG), can be shared for both static and measurement (OSIM) approaches. The results obtained from OSIM can be also used to understand the relationship between the distribution of the program's execution times and WCET throughout the execution paths.

By combining the power of both OTAWA and Lurette, the designers are now able to check both functional and timing properties of Lustre programs (can also apply to other synchronous programs). Our contributions include the follows:

1. Using automatic input generator (Lutin) to explorer more execution paths then conventional measurement approach.

2. Sharing the same facilities as the static analysis (OTAWA), the proposed approach make use of the CFG of the simulated programs, so that the occurrences (how many times a part has been executed) and execution times can be associated with the basic blocks (BB) and the edges between BBs. These information can be used as the feedback to the static analysis. Conventional measurement approach does not take execution paths into account.

This report is organized as follows: the organization of our framework is detailed in section 4. Section 5 presents the flow of the simulations. The measurements of the reactive programs and the requirements are described in Section 6. In section 7, how to use our approach to detect over-approximation is discussed. The results of the conducted experiments are explained in section 8, followed by section 9, the conclusion.

4 The organization of our framework

Figure 3 illustrates the overview of the framework, as well as the interconnections between the components in the framework. On the top level, the framework is based on Lurette, the inputs and outputs are established between (1) a stimuli generator modelled in Lutin, and (2) OSIM, a simulator to execute Lustre reactive programs on the simulated target processor.

OSIM is developed on top of OTAWA framework. OTAWA provides the infrastructures such as binary executable loading and decoding, so OSIM is able to access the reactive binary directly. OSIM consists of two parts, the instruction set simulator (ISS) and the structural simulator (SS). The ISS is used to determine the state of the simulated target processor and the SS is in charge of obtaining the cycle counts of the program execution.

The instruction set simulator (ISS) is automatically generated by a tool named GLISS2 [18], given the information of the instruction set architecture (ISA) is provided. The ISS maintains the state of the

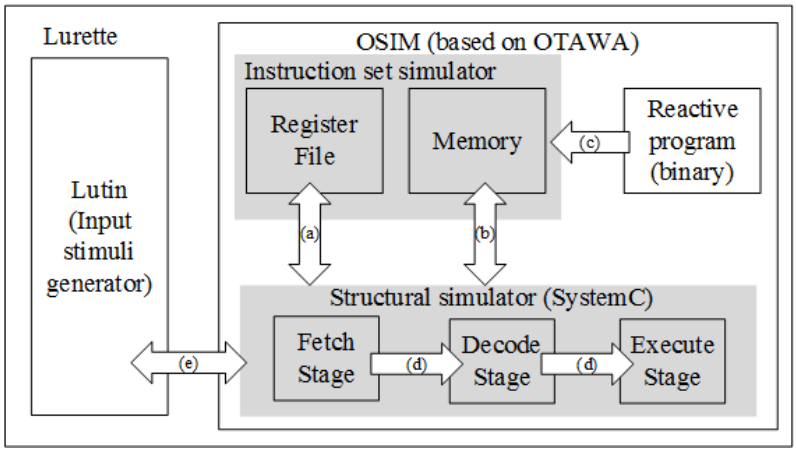

Figure 3: The structural of our proposed framework

processor in a register file (set of all registers in the processor) and the content of the memory accessed by the processor. The content of the memory is instantiated when needed to prevent over-uses of memory on the simulation host, i.e. even though 4GB of memory is available to the simulated target, it is wasteful to allocate the same size of memory spaces for the simulation. Instead, when an address is accessed by the processor, a segment of memory containing the accessed address is instantiated.

The ISS is implemented at the functional level instead of the cycle-accurate level. The operations of the processor are abstracted for better simulating performance. To demonstrate this, we use the instruction "add r1, r2, r2" as an example, which adds the contents of the register r1 and r2 and writes the results back to r1. In the RTL, the contents transferred from the register to the bus, as well as the computation occurs in the ALU, are described in a cycle accurate manner. In exchange of the performance in simulation, the functional ISS is used, where the results is computed by translate the instruction to a simple statement in C: R1 = R1 + R2, where R1 and R2 are part of the data structure representing the register file.

The SS is chosen to be modeled in SystemC [10], this is because (1) the notion of the clock cycles so that the behaviors of instruction moving from one processor stage to another at each cycle can be easily captured, (2) OSIM is based on OTAWA which is implemented with C++ and so is SystemC, this eases the integration of ISS and SS in OSIM, and (3) SystemC is suitable to model other hardware component so that OSIM can be extended by considering other possible hardware components such as memories with delay.

The SS is configured via XML files which describe the characteristics of the processor, such as the pipeline stages and the organization of the memories (i.e. the memory types with its corresponding addresses). Since we are interested in ARM7, the resulted SS has three pipeline stages: fetch, decode, and execute. The pipeline stages are only used to hold instructions to deduct how many cycles required for an instruction to complete. Hence the actual instruction decoding and execution are not performed in these stages, instead instructions are decoded and executed in the ISS. The execution stage is the most important stage in the SS. In contrast to fetch and decode stages which only hold the instruction for one cycle, the execute stage detects the type of the instruction and holds the instructions accordingly for a number of cycles until the instruction completes. In Figure 3 the arrows presents how different components exchange information:

(a) Interface to access the register file of the ISS, in particular, this enables the SS to obtain the correct instruction to fetch.
(b) The SS is able to read and write the content of memory to simulate the inputs and outputs from the memory mapped I/Os. The memory mapped I/Os are used as the means of communication between Lutin and the simulated programs.
(c) The ISS has access to the program binary and hence can provide the SS the instruction to fetch as described in (a).
(d) The instructions are transferred from one stage to its next stage when there is no blocking in the pipeline.

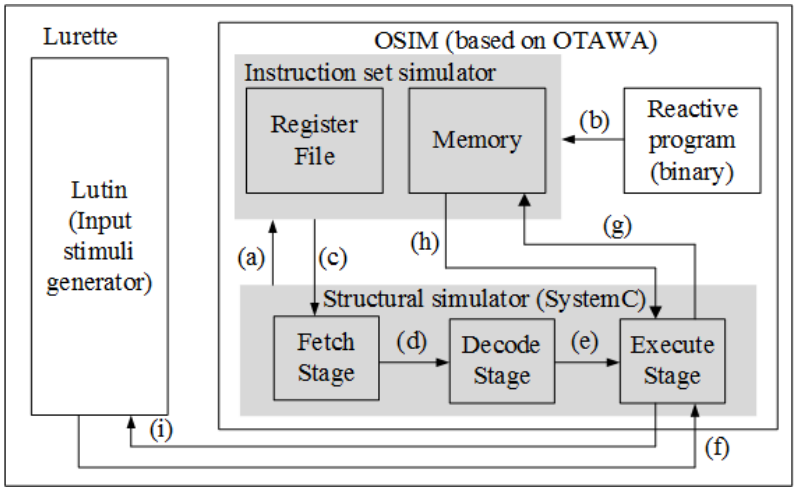

Figure 4: The simulation flow of the proposed framework

(e) The inputs from Lutin to the SS (in resp. to the outputs from the SS to Lutin) are implemented with TCP communications.

5 The flow of the simulation

Figure 4 illustrates how simulation is carried out internally. The simulation flow is described as the follows:

(1) The structural simulator (SS) send the request to obtain the instruction to fetch from the instruction set simulator (ISS) as (a).
(2) As (b) indicates, for obtaining the first instruction of the program, the ISS will provide it directly from the execution binary. For the later instructions, the ISS will execute the current instruction and determine the next instruction to provide.
(3) The SS now positions the received instruction to the Fetch stage as shown in (c).
(4) In the next cycle, the instruction in the fetch stage will be transferred to the decode stage, illustrated in (d). Meanwhile since the fetch stage is empty, it will request a new instruction as mentioned from (1) to (3).
(5) Similar to (4), the instruction in the decode stage will be put into the execution stage as (e). In the execution stage, the number of the cycles required to finish the instruction will be determined according to the opcode and the oprands of the instructions. For instance, a simple arithmetic instruction, e.g. ADD, will take only one cycle, while a branch instruction will take either one or three cycles according to the branching condition. When an instruction requires more than one cycle to complete, it stays in the execute stage. This will in turn stop the SS to request new instruction from the ISS. Once the instruction completes its cycles, the execute stage will be emptied, and the instruction will be transferred from the decode stage to execute stage, as well as from the fetch stage to the decode stage. In this case, new instruction will be requested from ISS as described in (1).
(6) When the simulated program requires the input from Lutin, the SS will read the input stimuli (currently implemented as a read operation with TCP socket) as shown in (f). Through the interface of ISS, the SS is able to transfer the input data to the specified memory address as shown in (g).
(7) Similarly when the output is generated from the simulated program, the SS will access the data from the memory (h) and pass it to Lutin (i).

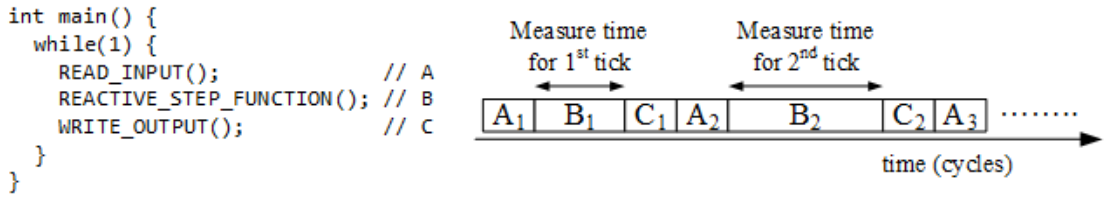

Figure 5: Execution of a reactive program

6 Measuring execution time of reactive programs and its I/Os

6.1 The execution model of reactive programs

A Reactive program described in Lustre is compiled to a reactive step function (or step function in short) in C (and other possible languages depending on the back-end of the code generation). The step function takes inputs from the environment, acts accordingly, and generates outputs back to the environment. Concurrency in the reactive programming languages are compiled away to a single thread code hence the problems of using thread [14] does not exist anymore. To support programming for real-time systems, a hypothesis is made that the worst-case execution time (WCET) of the step function must be known hence the needs of the timing analysis. The step function is included in an infinite loop so that it continuously reacts to the environment, as illustrated in Figure 5. We call each iteration of executing the step function a tick as Lustre has a notion of logical ticks. Each tick has different length, in the number of the processor cycles due to the inputs and the current state of the step function. Therefore in static analysis the WCET of the step function is taken into account.

To measure the length of the step function precisely, it is crucial to know when to start and stop the measurement. As step function is called in the main loop, and thanks to OTAWA 's facility to provide the CFG of the simulated program, it is possible to know the starting address and the end address of the function. We illustrate this with a Lustre program "*dependeur*" in Listing 1, and the corresponding CFG in Figure 6. OSIM only needs to know the name of the step function, i.e. dependeur_dependeur_step (provided with the argument -e), to conclude that the step function starts at address 0x8134 and ends at 0x8264.

Listing 1: The main loop of "*dependeur*" which calls the step function

```
00008270 <main>:
 ......                                          ; initializations
 827c:   mov   r3, #-536870912                   ; beginning of the while loop
 ......                                          ; reading inputs
 82ac:   bl    8134 <dependeur_dependeur_step>   ; step function
 82b0:   mov   r3, #-536870912                   ; start writing outputs
 ......                                          ; writing outputs continued
 82f0:   b     827c <main+0xc>                   ; back to the top of the while loop
```

6.2 The communications between Lutin, and OSIM

Figure 7 illustrates the communication between Lutin and OSIM within the framework. Such communication is achieved using the TCP on a specific port decided a priori. The inputs generated by Lutin will be passed to Lurette which forwards the inputs OSIM. The phases of communications are as follows:

(1) OSIM sends out the format of the I/O, which consists of the name and the type of each I/O, indicating OSIM is ready to receive inputs generated from Lutin, as shown in (a). Based on the I/O information, OSIM can determine the number of inputs and outputs and the corresponding addresses in the memory. Lutin received the I/O information (b) to confirm that OSIM is ready to communicate and ready to send out the generated inputs. The order of (a) and (b) at this handshaking phase is not important as TCP blocking read/write are used.

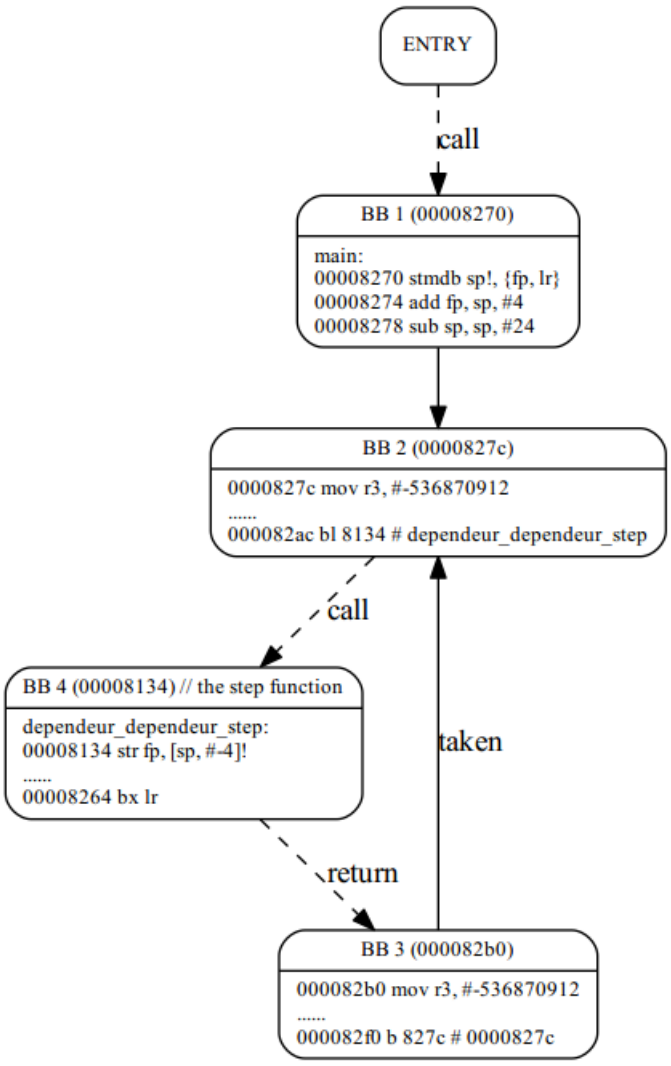

Figure 6: the CFG of the Lustre program *dependeur*

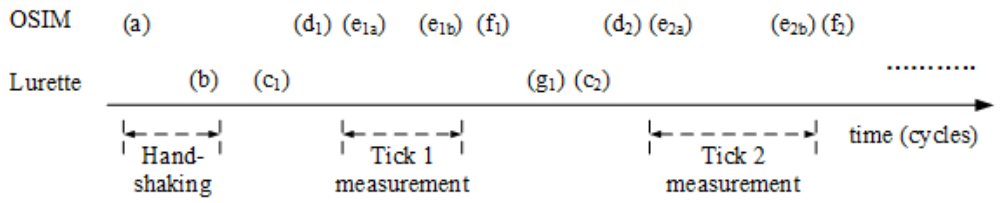

Figure 7: The communication between Lutin/Lurette and OSIM

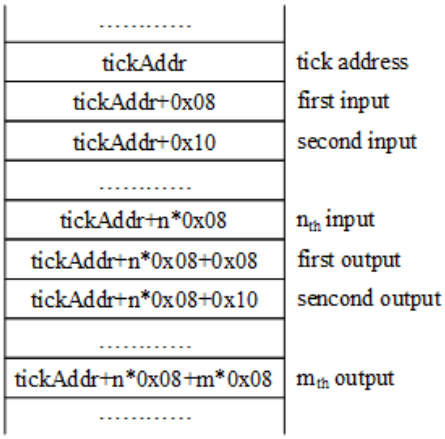

Figure 8: The memory mapping for the IOs of a simulated Lustre program

(2) Then Lutin sends out the first input (c_1) after the I/O information has been received.

(3) As shown in (d_1) ISS executes the instruction which loads the pre-specified tick address (TA) indicating the step function is ready to execute. TA is provided as an argument (-ta) to OSIM. SS then reads the inputs from (c_1) and stores them at the corresponding address of the memory content governed by the ISS. The organization of the addresses used by the simulated programs are illustrated in Figure 8.

(4) Once the simulation reaches (e_1a) which is the beginning of the reactive step function, the measurement starts. The measurement will stop to determine the number of cycles elapsed to execute the step function at (e_1b) where the last instruction of the step function completes.

(5) Once the step function completes, the SS reads the addresses mapped to outputs and send the obtained data to Lutin (f_1). After the outputs from the simulated program are received (g_1), Lutin sends out the inputs for the second tick (c_2). The simulation goes on in the same fashion in the later ticks (the loop of c_n to g_n).

(6) The simulation terminates once Lurette sends a termination message to OSIM at the end of the specified tick.

6.3 Memory mapping of inputs and outputs to the simulated programs

In order to have inputs from and outputs to Lutin to be successfully read/written by the simulated program, the main loop of the program must be conformed to the specified memory organization, i.e. the allocations of the tick address and the memory mapped I/Os. Listing 2 shows the source codes which implements the main loop for the program *dependeur*. The dedicated addresses for tick, inputs, and outputs are defined in lines 1-7. The corresponding memory mapping is illustrated in Figure 9. 8 bytes of memory is associated with each input and output for the simplicity and being able to handle 64-bit data types. The first address is always the tick address, followed by the inputs, and then the outputs. The first (tick) address can start at any reasonable memory location and the I/O addresses have to be in sequence as shown in the listing. The tick address has to be written at the beginning of the loop (line 11), the access of this address will trigger

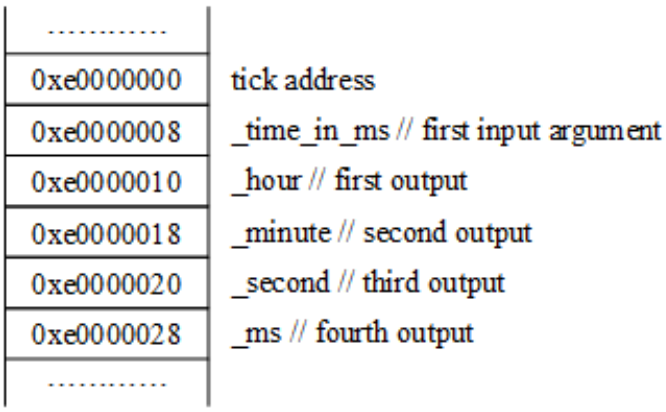

Figure 9: The memory organization of *dependeur*

OSIM to start measuring the execution cycles. The values assigned to the tick address does not affect the results. The inputs can be read from addresses as any ordinary readings from the memory mapped inputs (line 12) followed by calling the step function (line 13). The results of the step functions are assigned to the output addresses (lines 14-17) to conclude the execution loop.

Listing 2: The main loop of the Lustre reactive program *dependeur*

```
1  #include "dependeur_dependeur.h"
2  #define tickBegin    0xe0000000 // tick address
3  #define _time_in_ms  0xe0000008 // first input
4  #define _hour        0xe0000010 // first output
5  #define _minute      0xe0000018 // second output
6  #define _second      0xe0000020 // third output
7  #define _ms          0xe0000028 // fourth output
8  int main(){
9    _integer time_in_ms, hour, minute, second, ms;
10   while(1){                                                    // main loop
11     *((unsigned int*)tickBegin) = 0;                           // tick begins
12     time_in_ms = *((_integer *)_time_in_ms);                   // reading inputs
13     dependeur_dependeur_step(time_in_ms,&hour,&minute,&second,&ms); // step function
14     *((_integer *)_hour) = hour;                               // writing outputs
15     *((_integer *)_minute) = minute;
16     *((_integer *)_second) = second;
17     *((_integer *)_ms) = ms;
18   }
19   return 1;
20 }
```

7 Detecting over-approximation of timing analysis

Since OSIM keeps track of the control flow graph (CFG) of the simulated program. Information such as how many times (occurrences) a basic block has been executed, the number of execution cycles, and the total execution cycles of a basic block are provided alongside the simulation. Similarly, information of the edges are also provided in the same fashion. This is useful to observe the possible over-approximation of the analysis. For obtaining the statistics, please refer to section 10.1. For example, an edge E_1_2 between basic blocks BB_1 and BB_2 are passed 5 times during the simulation. The maximum number of cycle among the 5 times is 6 cycles, and the total amount of cycles on E_1_2 is 24 cycles. The WCET for the measurement approach for this edge will be 6*5 = 30 cycles, hence the over-approximation of 4 cycles.

Table 1: bsort100 – the comparisons between static and measurement analysis

	A	B	C	D	E	F	G	H	I
1	#id	Occurrences (ILP)	max_cycle (OTAWA)	Occurrences (OSIM)	max_cycle (OSIM)	WCET (OTAWA)	WCET (OSIM_MEASURE)	WCET (OSIM_CALC)	Over Estimation (static/dynamic) times
2	e0_1_main	1	10	1	11	10	11	11	0.91
3	e1_4_main	1	8	1	6	8	6	6	1.33
4	e2_7_main	1	13	1	13	13	13	13	1.00
5	e3_16_main	1	0	0	0	0	0	0	-
6	e4_5_main	1	5	1	7	5	7	7	0.71
7	e5_5_main	100	7	99	7	700	691	693	1.01
8	e5_6_main	1	1	1	3	1	3	3	0.33
9	e6_2_main	1	4	1	4	4	4	4	1.00
10	e7_8_main	1	6	1	6	6	6	6	1.00
11	e8_9_main	100	17	99	17	1700	1683	1683	1.01
12	e9_10_main	0	2	1	2	0	2	2	0.00
13	e9_11_main	9901	4	4949	4	39604	10094	19796	3.92
14	e10_12_main	100	2	99	4	200	394	396	0.51
15	e10_13_main	0	11	0	0	0	0	0	-
16	e11_10_main	100	4	98	2	400	196	196	2.04
17	e11_9_main	9801	15	4851	17	147015	82465	82467	1.78
18	e12_14_main	1	1	1	3	1	3	3	0.33
19	e12_15_main	99	4	98	4	396	392	392	1.01
20	e13_3_main	1	7	1	7	7	7	7	1.00
21	e14_13_main	1	11	1	11	11	11	11	1.00
22	e15_8_main	99	6	98	6	594	588	588	1.01
23	sum					190675	96576	106284	1.97

The static analysis from OTAWA will use the maximum cycle counts of the edge e, for the sake of simplicity we use 6 cycles. The occurrences of the edge is computed by solving the ILP (integer linear programming). The solution provided by the ILP may provide some value such as 7 for example. The WCET for the edge e is then 6*7 = 42 cycles, and therefore the over-approximation of 18 cycles.

From this example we can see that the over-approximation comes from two places: (1) the worst case cycle count is used, and (2) the computed occurrences of the edge. Such information can also be used to see if the generated ILP formula are correct, also can be used to understand the behaviors of the simulated program which might be relevant to the analysis method hence the over-approximations.

To demonstrate, we use the bsort100 (bubble sort of 100 numbers) example from the Mälardalen benchmarks [7]. The statistics are shown in Table 1. To ease the explanation, the partial control flow graph (CFG) of bsort100 is provided as Figure 10. The column A in Table 1 indicates the name of the edge, for example, e0_1_main represents the edge between the block ENTRY and BB 1. The static analysis of OTAWA makes use of integer linear programming (ILP) to compute the occurrences of the edge, as illustrate in column B. Column C indicates the worst-case cycle-count associated with each edge. The WCET of each edge is its occurrences multiply by its worst-case cycle-count as resulted in column F, in this case the WCET obtained from the static analysis is 190,675 cycles. Similarly, column D and column E record the occurrence and worst-case cycle-counts obtained by using OSIM. OSIM also provides the accumulated worst-case cycle-counts for each edge, as shown in column G, and result 96,576 cycles as the WCET from the measurement approach. We provide the WCET estimation based on the results from OSIM on column H. The values are computed in the similar fashion: the product of the occurrence times the worst-case cycle counts obtained by using OSIM. To observe the difference between the static and measurement approach, we provide the ratio of them on column I.

By observing Table 1, the occurrence and worst-case cycle-counts are differed due to the following reasons:

(1) The differences in worst-case cycle counts: this is due to the strategy of counting the number of cycles per basic block (BB). The cycles can be counted from the first instruction is fetched, till the

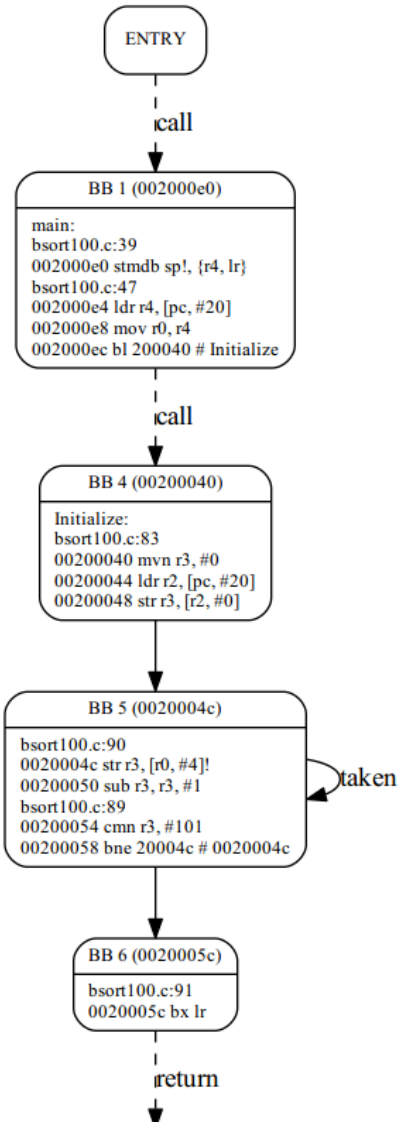

Figure 10: The partial CFG of bsort100

last instruction completes its execution. Another approach can be counting the number of cycles from the completion of the last instruction of the previous BB, till the last instruction of current BB finishes its execution. Even though the strategies are different, the effects can be either minimal to none as it is just a matter of interpretation. For example, from Figure 10 we can see that the execution path is ENTRY-BB1-BB4-BB5 which only occurs once for both static and measurement analysis. The WCET on this path for the static approach is 10 (from C:2) + 8 (C:3) + 5 (C:6) = 23, while for measurement approach is 11 (E:2) + 6 (E:3) + 7 (E:6) = 23 which equals to the former case.

(2) The minor differences in the occurrences: for rows 7, 11, 14, 16, 19, and 22, we can see the difference of one or two occurrences in column B (static) and column D (measurement). This is due to the static analysis relied on FFX files [2] to determine the loop bounds, but the CFGs of the executables may not be a full correspondence to the FFX files. Listing 3 is the partial FFX for the first loop of bsort100, and the corresponding sources are presented in Listing 4. The loop is bounded to 100 iterations (the value defined by NUMELEMS), hence the maxcount="100" (line 7) of Listing 3. By referring to Figure 11, the loop body is associated with BB 5, which results that the occurrence of edge e4_5_main to be

1 and e5_5_main to be 99. This can be due to that the binary is optimized so that the loop body is presented in a do-while statement fashion (the condition of the first iteration is not checked as in for-loop). Such difference could lead to larger over-approximation if the body of the loop is considerably huge.

Listing 3: The flow fact for the first loop in *bsort100*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<flowfacts>
   <function name="main" executed="true" extern="false">
      <call name="Initialize" numcall="3" line="47" source="bsort100.c"
            executed="true" extern="false">
         <function name="Initialize">
            <loop loopId="1" line="89" source="bsort100.c" exact="true"
                  maxcount="100" totalcount="100">
            </loop>
            ..................
```

Listing 4: The source for the first loop in *bsort100*

```
for (Index = 1; Index <= NUMELEMS /* 100 */; Index ++)
    Array[Index] = Index*fact * KNOWN_VALUE;
}
```

(3) The major differences in the occurrences: for rows 13 and 17, the differences of occurrences between the static (9901 and 9801 times) and measurements (4949 and 4851 times) approach are significant. Such difference is amplified because of the worst-case cycle-counts of the edges. The edges are associated to the main body of the bubble sort. The number of iterations of the bubble sort heavily depending on how the original set is formed. It appears that the given set does not require the maximum possible iterations to be sorted, hence the over-estimation from the static analysis. Such over-estimation is necessary to safely predict the WCET of the program, however the measurement approach provides the insights the possible sources of differences of estimation times.

8 Experiments

8.1 Case study: mode3x2

The modes3x2 [19] is an example of multi-mode programs which processes data in different behaviors according to the current operation mode as illustration in Figure 12. The program is controlled by two Boolean inputs *onoff* and *toggle*, to process the *data* (integer input) and to generate an output named *out*. It is important to note that the two Boolean inputs are exclusive with each other, i.e. only one of them can be true. This has to be taken into account when designing the input stimuli with Lutin otherwise the program will have unexpected behaviors. To demonstrate this, the WCET of both our approach and static analysis from OTAWA are illustrated in Figure 13. To show the impact of correct designed input stimuli, we use a counter-example such that both *onoff* and *toggle* can be true at the same time, and the results are shown in Figure 14. It is noticed that, some of the measurements exceeded the static analysis (circled in green). This indicates having the correct set of inputs is crucial and yet it is error prone to provides inputs manually by designers. This again shows the usefulness of having inputs to be generated automatically by tools such as Lutin in our proposed method.

We simulated nodes3x2 on a 2.8 GHz Core i7 machine, the simulation only used one processor. It took 47 seconds to simulate the step function of modes3x2 for 1000 ticks for a total amount of 2,437,769 cycles (2,565,891 cycles for the whole program including the starting of the run-time, main function, etc.). The WCET that we obtain is of 2689 cycles at the 318th tick (this may vary according to the generated inputs).

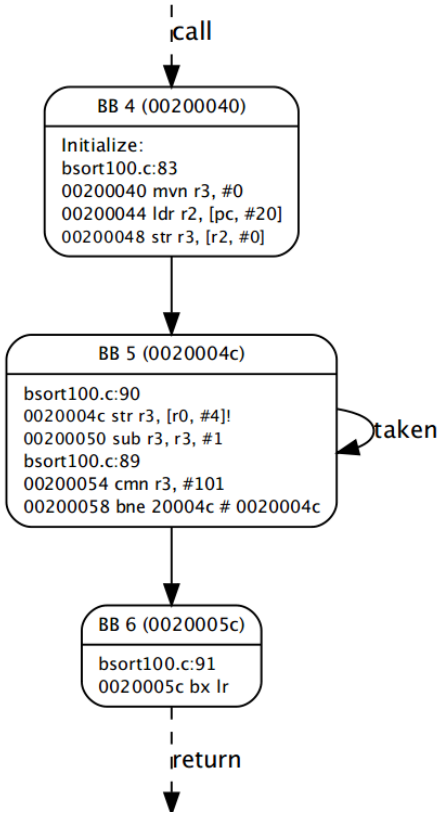

Figure 11: The partial CFG corresponding to the first loop in *bsort100*

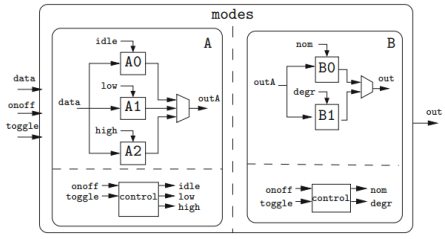

Figure 12: The modes3x2 example

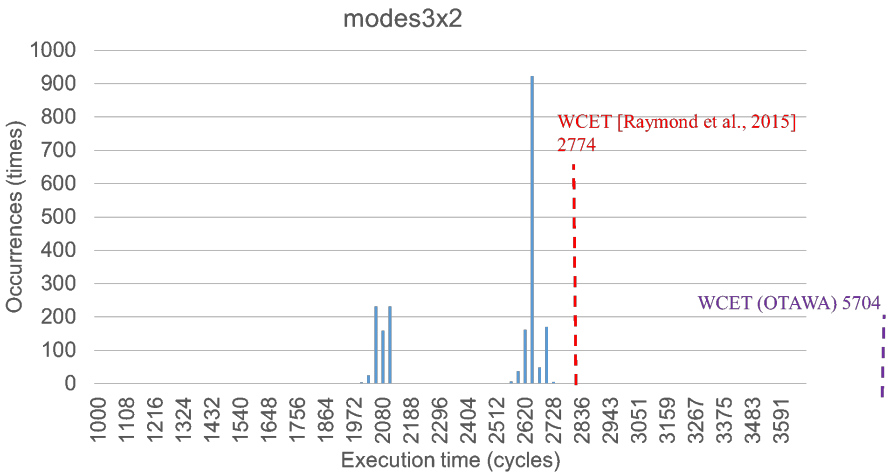

Figure 13: The WCET of our approach (blue), static analyses (red and purple)

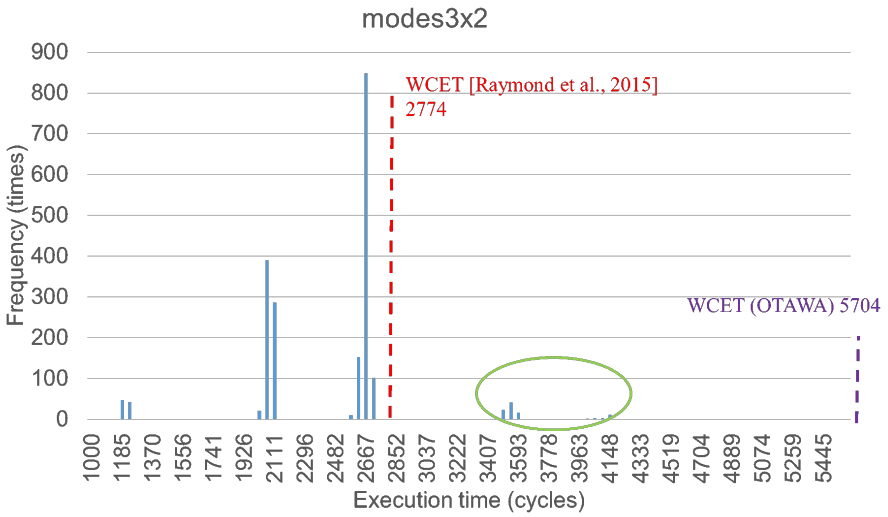

Figure 14: The unexpected behaviors due to wrongly designed input stimuli

Table 2: The comparisons of WCET from different approaches

Method	Cycles
OSIM	2689
OWCET (OTAWA)	5704
OWCET (Raymond et al., 2015)	2774

The comparisons of the results as shown in Table 2. Table 3 shows the details of the occurrences of the edges for the corresponding WCET. Our approaches and the approach in [19] share the same execution traces (all the values in the column "diff paths OSIM vs 2015 are 0, which means no differences) of the simulated program. The general static analysis did not consider the infeasible paths, and took different paths (as colored in blue) to have higher WCET. The rows colored yellow and orange are the major contribution of the different WCETs between our approach and [19]. These two rows are corresponding to the function calls A2 and B1 which use *MUL* instructions that have a variable execution time according to the operated data. The worst-case time of the MUL instructions did not happen in the simulation but is considered in the static analysis (in the WCET columns).

Table 3: The comparisons of paths and WCET for different approaches

#id	max cycle (OSIM)	occurrences (OSIM)	max cycle (OWCET)	occurrences (OWCET)	occurrence (OWCET 2015)	Diff paths (OSIM vs OWCET)	Diff paths (OSIM vs 2015)	Diff cycles (OSIM vs 2015) %
e0_1_modes3x2_step	16	1	15	1	1	0	0	1.2
e1_2_modes3x2_step	0	0	4	0	0	0	0	0.0
e1_3_modes3x2_step	8	1	10	1	1	0	0	-2.4
e10_12_modes3x2_step	0	0	10	0	0	0	0	0.0
e11_12_modes3x2_step	10	1	8	1	1	0	0	2.4
e12_13_modes3x2_step	0	0	4	0	0	0	0	0.0
e12_14_modes3x2_step	8	1	10	1	1	0	0	-2.4
e13_15_modes3x2_step	0	0	45	0	0	0	0	0.0
e14_15_modes3x2_step	45	1	43	1	1	0	0	2.4
e15_16_modes3x2_step	0	0	4	0	0	0	0	0.0
e15_17_modes3x2_step	8	1	10	1	1	0	0	-2.4
e16_18_modes3x2_step	0	0	28	0	0	0	0	0.0
e17_18_modes3x2_step	26	1	26	1	1	0	0	0.0
e18_19_modes3x2_step	8	1	6	1	1	0	0	2.4
e18_20_modes3x2_step	0	0	7	0	0	0	0	0.0
e19_21_modes3x2_step	7	1	7	1	1	0	0	0.0
e2_4_modes3x2_step	0	0	18	0	0	0	0	0.0
e20_21_modes3x2_step	0	0	5	0	0	0	0	0.0
e21_22_modes3x2_step	0	0	10	1	0	-1	0	0.0
e21_23_modes3x2_step	7	1	7	0	1	1	0	0.0
e22_84_modes3x2_step	0	0	910	1	0	-1	0	0.0
e23_24_modes3x2_step	0	0	6	0	0	0	0	0.0
e23_25_modes3x2_step	8	1	10	1	1	0	0	-2.4
e24_26_modes3x2_step	0	0	28	0	0	0	0	0.0
e25_26_modes3x2_step	28	1	26	1	1	0	0	2.4
e26_27_modes3x2_step	0	0	6	1	0	-1	0	0.0
e26_28_modes3x2_step	5	1	7	0	1	1	0	-2.4
e27_29_modes3x2_step	0	0	7	1	0	-1	0	0.0
e28_29_modes3x2_step	5	1	5	0	1	1	0	0.0
e29_30_modes3x2_step	12	1	10	1	1	0	0	2.4
e29_31_modes3x2_step	0	0	7	0	0	0	0	0.0
e3_4_modes3x2_step	18	1	16	1	1	0	0	2.4
e30_85_modes3x2_step	1487	1	1562	1	1	0	0	-88.2
e31_32_modes3x2_step	8	1	6	0	1	1	0	2.4
e31_33_modes3x2_step	0	0	10	1	0	-1	0	0.0
e32_34_modes3x2_step	10	1	10	0	1	1	0	0.0
e33_34_modes3x2_step	0	0	8	1	0	-1	0	0.0
e34_35_modes3x2_step	0	0	4	0	0	0	0	0.0
e34_36_modes3x2_step	8	1	10	1	1	0	0	-2.4
e35_37_modes3x2_step	0	0	7	0	0	0	0	0.0
e36_37_modes3x2_step	5	1	5	1	1	0	0	0.0
e37_38_modes3x2_step	8	1	6	1	1	0	0	2.4
e37_39_modes3x2_step	0	0	7	0	0	0	0	0.0
e38_40_modes3x2_step	7	1	7	1	1	0	0	0.0
e39_40_modes3x2_step	0	0	5	0	0	0	0	0.0
e4_5_modes3x2_step	0	0	6	0	0	0	0	0.0
e4_6_modes3x2_step	8	1	10	1	1	0	0	-2.4
e40_41_modes3x2_step	0	0	6	1	0	-1	0	0.0
e40_42_modes3x2_step	5	1	7	0	1	1	0	-2.4
e41_43_modes3x2_step	0	0	7	1	0	-1	0	0.0
e42_43_modes3x2_step	7	1	5	0	1	1	0	2.4
e43_44_modes3x2_step	0	0	6	1	0	-1	0	0.0
e43_45_modes3x2_step	5	1	7	0	1	1	0	-2.4
e44_46_modes3x2_step	0	0	10	1	0	-1	0	0.0
e45_46_modes3x2_step	10	1	8	0	1	1	0	2.4
e46_47_modes3x2_step	0	0	4	0	0	0	0	0.0
e46_48_modes3x2_step	8	1	10	1	1	0	0	-2.4
e47_49_modes3x2_step	0	0	21	0	0	0	0	0.0
e48_49_modes3x2_step	21	1	19	1	1	0	0	2.4

#id	max cycle (OSIM)	occurrences (OSIM)	max cycle (OWCET)	occurrences (OWCET)	occurrence (OWCET 2015)	Diff paths (OSIM vs OWCET)	Diff paths (OSIM vs 2015)	Diff cycles (OSIM vs 2015) %
e49_51_modes3x2_step	7	1	7	0	1	1	0	0.0
e5_7_modes3x2_step	0	0	7	0	0	0	0	0.0
e50_86_modes3x2_step	0	0	1438	1	0	-1	0	0.0
e51_52_modes3x2_step	0	0	6	0	0	0	0	0.0
e51_53_modes3x2_step	8	1	10	1	1	0	0	-2.4
e52_54_modes3x2_step	0	0	10	0	0	0	0	0.0
e53_54_modes3x2_step	10	1	8	1	1	0	0	2.4
e54_55_modes3x2_step	0	0	4	0	0	0	0	0.0
e54_56_modes3x2_step	8	1	10	1	1	0	0	-2.4
e55_57_modes3x2_step	0	0	21	0	0	0	0	0.0
e56_57_modes3x2_step	19	1	19	1	1	0	0	0.0
e57_58_modes3x2_step	8	1	6	1	1	0	0	2.4
e57_59_modes3x2_step	0	0	7	0	0	0	0	0.0
e58_87_modes3x2_step	466	1	476	1	1	0	0	-11.8
e59_60_modes3x2_step	8	1	6	0	1	1	0	2.4
e59_61_modes3x2_step	0	0	10	1	0	-1	0	0.0
e6_7_modes3x2_step	7	1	5	1	1	0	0	2.4
e60_62_modes3x2_step	10	1	10	0	1	1	0	0.0
e61_62_modes3x2_step	0	0	8	1	0	-1	0	0.0
e62_63_modes3x2_step	0	0	4	0	0	0	0	0.0
e62_64_modes3x2_step	8	1	10	1	1	0	0	-2.4
e63_65_modes3x2_step	0	0	7	0	0	0	0	0.0
e64_65_modes3x2_step	5	1	5	1	1	0	0	0.0
e65_66_modes3x2_step	8	1	6	1	1	0	0	2.4
e65_67_modes3x2_step	0	0	7	0	0	0	0	0.0
e66_68_modes3x2_step	7	1	7	1	1	0	0	0.0
e67_68_modes3x2_step	0	0	5	0	0	0	0	0.0
e68_69_modes3x2_step	0	0	6	1	0	-1	0	0.0
e68_70_modes3x2_step	5	1	7	0	1	1	0	-2.4
e69_71_modes3x2_step	0	0	15	1	0	-1	0	0.0
e7_8_modes3x2_step	0	0	10	1	0	-1	0	0.0
e7_9_modes3x2_step	7	1	7	0	1	1	0	0.0
e70_71_modes3x2_step	15	1	13	0	1	1	0	2.4
e71_88_modes3x2_step	20	1	20	1	1	0	0	0.0
e72_73_modes3x2_step	14	1	14	1	1	0	0	0.0
e72_74_modes3x2_step	0	0	7	0	0	0	0	0.0
e73_74_modes3x2_step	7	1	5	1	1	0	0	2.4
e74_75_modes3x2_step	0	0	14	1	0	-1	0	0.0
e74_76_modes3x2_step	47	1	49	0	1	1	0	-2.4
e75_76_modes3x2_step	0	0	47	1	0	-1	0	0.0
e76_77_modes3x2_step	14	1	14	1	1	0	0	0.0
e76_78_modes3x2_step	0	0	7	0	0	0	0	0.0
e77_78_modes3x2_step	7	1	5	1	1	0	0	2.4
e78_79_modes3x2_step	0	0	14	1	0	-1	0	0.0
e78_80_modes3x2_step	35	1	35	0	1	1	0	0.0
e79_80_modes3x2_step	0	0	33	1	0	-1	0	0.0
e8_83_modes3x2_step	0	0	512	1	0	-1	0	0.0
e80_81_modes3x2_step	0	0	14	1	0	-1	0	0.0
e80_82_modes3x2_step	26	1	27	0	1	1	0	-1.2
e81_82_modes3x2_step	0	0	25	1	0	-1	0	0.0
e82_89_modes3x2_step	0	1		1	1	0	0	0.0
e83_9_modes3x2_step	0	0	7	1	0	-1	0	0.0
e84_23_modes3x2_step	0	0	7	1	0	-1	0	0.0
e85_31_modes3x2_step	5	1	7	1	1	0	0	-2.4
e86_51_modes3x2_step	0	0	7	1	0	-1	0	0.0
e87_59_modes3x2_step	5	1	7	1	1	0	0	-2.4
e88_72_modes3x2_step	79	1	81	1	1	0	0	-2.4
e9_10_modes3x2_step	0	0	6	0	0	0	0	0.0
e9_11_modes3x2_step	8	1	10	1	1	0	0	-2.4
WCET		2689		5704	2774			

Table 4: The difference between static and measurement approach for Lustre v6 benchmarks

Example	OWCET	OSIM	Diff	Example	OWCET	OSIM	Diff	Example	OWCET	OSIM	Diff
aa	149	149	0	compteur	171	169	2	mouse1	995	985	10
call02	59	59	0	contractForElementSelectionInArr	130	128	2	test_diese	192	182	10
consensus	845	845	0	minmax1	64	62	2	toolate	995	985	10
consensus2	845	845	0	minmax2	99	97	2	ck2	289	278	11
count	35	35	0	mm	64	62	2	cminus	655	644	11
cst	56	56	0	mm1	62	60	2	test	607	596	11
dependeur	154	154	0	mm22	73	71	2	flo	433	421	12
dependeur_struct	150	150	0	mm3	92	90	2	mouse	1346	1334	12
fresh_name	83	83	0	pre_x	431	429	2	mouse2	1346	1334	12
impl_priority	62	62	0	rs	469	467	2	PCOND1	481	467	14
nc1	88	88	0	sample_time_change	368	366	2	call07	243	229	14
nc10	619	619	0	simpleRed	173	171	2	lucky	1448	1434	14
nc2	114	114	0	test_arrow	300	298	2	redoptest	583	569	14
nc3	190	190	0	test_condact	428	426	2	stopwatch	1156	1141	15
nc4	190	190	0	trivial2	369	367	2	mappredef	408	392	16
nc5	173	173	0	xx	116	114	2	deSimone	2984	2964	20
nc6	191	191	0	yyy	116	114	2	mapinf	558	538	20
nc7	225	225	0	COUNTER	191	188	3	minmax6	791	771	20
nc8	884	884	0	STABLE	263	260	3	alias	690	669	21
nc9	604	604	0	clock_ite	154	151	3	pipeline	2295	2274	21
noAlarm	26	26	0	dep	436	433	3	uu	1450	1429	21
node_caller1	182	182	0	double_delay	420	417	3	iter	1533	1511	22
nodeparam	112	112	0	initial	450	447	3	test_boolred	254	232	22
noeudsIndependants	31	31	0	test_merge	161	158	3	X6	1100	1071	29
notTwo	37	37	0	call04	249	245	4	rediter	1321	1291	30
param_node	72	72	0	ck4	127	123	4	access	1752	1721	31
param_node2	73	73	0	ck5	394	390	4	PCOND	1092	1059	33
param_node3	89	89	0	ck7	148	144	4	plus	669	633	36
param_node4	114	114	0	clock1_2ms	473	469	4	predef02	1112	1070	42
param_struct	89	89	0	X2	150	145	5	mapiter	1576	1528	48
struct0	27	27	0	argos	847	842	5	SOURIS	5945	5896	49
t1	865	865	0	cpt	187	182	5	SOURIS_v6	5945	5896	49
t2	586	586	0	enum0	125	120	5	calculs_max	5611	5561	50
testCA	154	154	0	redIf	278	273	5	iterate	3938	3888	50
test_const	27	27	0	CURRENT	136	130	6	modes3x2_v4	1063	1006	57
test_node_expand	108	108	0	SWITCH	272	266	6	predef03	542	484	58
titi	49	49	0	SWITCH1	181	175	6	modes3x2_v3	1191	1127	64
ts01	24	24	0	TIME_STABLE1	404	398	6	carV2	4143	4070	73
ts04	24	24	0	ck3	212	206	6	struct_with	231	151	80
tuple	43	43	0	minmax3	254	248	6	over3	2074	1990	84
type_decl	33	33	0	TIME_STABLE	440	433	7	ply01	2074	1990	84
zzz	55	55	0	bob	549	542	7	predef01	2074	1990	84
zzz2	55	55	0	mouse3	571	564	7	exclusion	1386	1297	89
EDGE	162	161	1	poussoir	660	652	8	modes3x2_v2	2372	2255	117
FALLING_EDGE	201	200	1	test_enum	236	228	8	test_struct	422	267	155
after	159	158	1	activation_ec	657	648	9	bad_call03	945	768	177
ex	226	225	1	test_node_expand2	271	262	9	map	1099	847	252
followed_by	155	154	1	X	596	586	10	overload	941	689	252
long_et_stupide_nom_de_noeud	155	154	1	bred	146	136	10	test_map	1099	847	252
trivial	193	192	1	bred_lv4	146	136	10	sample_time_change_MainNode	3439	3063	376
v1	40	39	1	minmax4	410	400	10	FillFollowedByRed	5330	3648	1682
bascule	474	472	2	minmax4_bis	410	400	10				

8.2 Evaluations of Lustre v6 benchmarks

We conducted a series of static and measurement analysis on the benchmark of Lustre v6. The benchmark consists of 173 programs which uses different features and combinations of the features. Even though the measurement approach can obtain the WCET for all 173 programs, the static approach was made successfully on 155 of them. They results are detailed in Table 4, which shows the WCET from the static and measurement approach, as well as the difference between them. Table 5 shows the statistics of the comparisons. More than one-fourth (27.7%) of examples have the same WCET on both static and measurement analysis, and 94.2% are within the less than 10% range. This shows the static analysis is generally accurate, especially over control-dominated programs. This is because the benchmark does not includes examples which have huge computational differences between different paths within the program.

Table 5: Statistics of simulated programs in Lustre v6 benchmark

	Examples	%	Accumulative %
Same	43	27.7	27.7
(0%, 1%]	27	17.4	45.2
(1%, 2%]	29	18.7	63.9
(2%, 3%]	17	11.0	74.8
(3%, 4%]	15	9.7	84.5
(4%, 5%]	6	3.9	88.4
(5%, 6%]	5	3.2	91.6
(6%, 7%]	3	1.9	93.5
(8%, 9%]	1	0.6	94.2
(10%, 20%]	3	1.9	96.1
(20%, 30%]	3	1.9	98.1
(30%, 40%]	3	1.9	100.0
Total	155	100.0	

9 Conclusions

In this report we present a framework of integrating high-level testing environment for reactive programs (Lurette) with measurement-based WCET tools (OSIM). The integration of Lurette and OSIM provides the execution times of reactive programs with different input scenarios. With this framework, the designers are able to verify both functional and timing aspects of the reactive programs.

OSIM enables the users to observe the distributions of the execution times of not only reactive programs but also other programs designed for real-time systems. Facilitated by OTAWA, OSIM is able to detect the over-approximations made from static analysis by referring the occurrences and worst-case time associated with basic blocks and edges between them.

10 Appendix

10.1 Statistics of measurement and detecting of over-approximation for non-reactive programs

Obtaining the statistics of the occurrences, the worst-case cycles, and the total amount of cycles of basic blocks (BBs) and edges between BBs. The statistics will be stored in the provide argument to *otawa::osim-lpc2138::TRACE_INFO*.

osim-lpc2138 -p proc.xml -m memory.xml prog.elf –add-prop otawa::osim-lpc2138::TRACE_INFO=trace.txt

The content of the statics will be in the following format:

```
id,          occurrence,  max_cycle,  total_cycles
x0_main,      0,           0,          0
e0_1_main,    1,           18,         18
x1_main,      1,           18,         18
e1_3_main,    1,           154,        154
x2_main,      1,           9,          9
e2_4_main,    0,           0,          0
x3_main,      1,           154,        154
e3_2_main,    1,           9,          9
x4_main,      0,           0,          0
```

The naming convention of the ids is the same as in the ILP formula generated by *owcet*. For example, x0_main indicates the first (with index = 0) basic block of the simulated program. e0_1_main indicates the edge between the first basic block and the second basic block.

The occurrences and WCET on edges in the static analysis can be obtained from the facility of ILP generation and solution, it is require to use the processor in the script of owcet (lpc2138.osx):

```
1 <script>
2       ......
3       <step processor="otawa::ilp::Output"/>
4 </script>
```

The ILP equations can be output to a file by the following command:

owcet.arm prog.elf –add-prop otawa::ilp::OUTPUT_PATH=prog.ilp

To solve the ilp:

lp_solve < prog.ilp > prog.sol

The first line of the ILP equations contains the worst-case cycle-counts of the edges as the coefficients, e.g. max: 4 e12_15_main + e12_14_main + 11 e10_13_main + 11 e14_13_main + 2 e10_12_main + 4 e9_11_main + 2 e9_10_main + 4 e11_10_main + 17 e8_9_main + 15 e11_9_main + 6 e7_8_main + 6 e15_8_main + 13 e2_7_main + e5_6_main + 5 e4_5_main + 7 e5_5_main + 8 e1_4_main + 7 e13_3_main + 4 e6_2_main + 10 e0_1_main;

Note that if the edge is without coefficient means that the coefficient is 1, e.g. for edge e12_14_main.

The occurrences of the edges to maximize the WCET are stored in the sol file:

e12_15_main	99
e12_14_main	1
e10_13_main	0
e14_13_main	1
e10_12_main	100
e9_11_main	9901
e9_10_main	0
e11_10_main	100
e8_9_main	100
e11_9_main	9801
e7_8_main	1
e15_8_main	99
e2_7_main	1
e5_6_main	1
e4_5_main	1
e5_5_main	100
e1_4_main	1
e13_3_main	1
e6_2_main	1
e0_1_main	1

owcet.arm modes3x2_infini.elf modes3x2_step –add-prop otawa::ilp::OUTPUT_PATH=modes3x2.ilp

10.2 Command lines and options for running OSIM with Lurette to simulate reactive programs

OSIM and Lurette are running as two different processes communicating with each other through TCP. To provide the input stimuli, the input generator described in Lutin must be available. For example, to simulate modes3x2 as described in the experimental section, OSIM can be launched as illustrated in Listing 5. The configuration of the simulated processor and its memory hierarchy are provided as lines 2-3. Line 4 (-ul,

stands for using lurette) indicates the simulation will be an interaction with Lurette. The I/O format file is provide through the -iol (stands for I/O list) argument, which is used at the phase (a) in Figure 7. An example of the IO format file is shown as Listing 6 for modes3x2. OSIM can generate the statistics for each describing the occurrences, WCET of each BB and edges, and the summation of the cycles as described in section 10.1. This is realized through adding the TICK_TRACE_INFO (respective to TRACE_INFO shown in the previous section, available from the version 2016-April-10) as shown in line 6. The string "logs/tick" specifies the format of the generated file names, i.e. logs/tick_1 for the first tick. The -cl option writes the cycle measurements of ticks in the specified file (line 7). The reactive step function is pointed by the -e argument as shown in line 8. Line 9 indicates the binary executable to measure, while line 10 is used to set the TCP port to communicate with Lurette.

Listing 5: The command line to launch OSIM to simulate modes3x2

```
1  osim-lpc2138 \
2      -p ${OSIM_PATH}/proc.xml \
3      -m ${OSIM_PATH}/memory.xml \
4      -ul 1 \
5      -iol modes3x2.io \
6      --add-prop otawa::osim-lpc2138::TICK_TRACE_INFO=logs/tick \
7      -cl modes3x2.cycles \
8      -e modes3x2_step \
9      obj/modes3x2.elf \
10     -lp 2002
```

Listing 6: The IO format file loaded by OSIM and sent to Lurette

```
1  #inputs x:int on_off:bool toggle:bool
2  #outputs res:int
```

Similarly, Lurette requires a set of argument in order to correctly couple with OSIM, as shown in Listing 7. Because OSIM (acts as SUT, system under test) communicates with Lurette through TCP (of the local machine, can be remote too), the address and the port are specified as line 2. The uses of Lutin is detailed in line 3, where the name of the Lutin file (modes3x2_input.lut) and the main entry of the input generator (modes3x2_input) are specified in line 3. Line 4 indicates the number of the logical ticks (in the Lustre fashion) to simulate. -go option in line directs Lurette to start the simulation right away (line 5). -no-sim2chro in line 6 says Lurette will not launch the graphical interface to show the simulation results (optional, but useful for batch processing the benchmarks).

Listing 7: The command line to launch Lurette

```
1  lurettetop_exe \
2      -rp sut:socket:127.0.0.1:2002 \
3      -rp "env:lutin:modes3x2_input.lut:modes3x2_input" \
4      -l 1000 \
5      -go \
6      --no-sim2chro
```

At the end of the simulation, OSIM will print out something like:

WCET happens at tick 318 for 2689

The user can open the file logs/tick_317 to investigate the execution statistics of the tick. It uses the same naming conventions as other tools, e.g. owcet.arm as well as w7. This enables us to find out the differences in occurrences of paths to understand the over-approximation of the static analysis.

References

[1] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003. 1

[2] Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Ffx: A portable wcet annotation language. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 91–100. ACM, 2012. 7

[3] Hugues Cassé, Haluk Ozaktas, and Christine Rochange. A Framework to Quantify the Overestimations of Static WCET Analysis. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, 2015. 2

[4] Hugues Cassé and Pascal Sainrat. Otawa, a framework for experimenting wcet computations. In *3rd European Congress on Embedded Real-Time Software*, 2006. 1

[5] Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 5, 2005, Palma de Mallorca, Spain*, 2005. 2

[6] Jorge Garrido, Daniel Brosnan, Juan Antonio de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, pages 81–90, 2012. 2

[7] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010. 7

[8] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. 1

[9] Erik Yu-Shing Hu, Andy J. Wellings, and Guillem Bernat. Deriving java virtual machine timing models for portable worst-case execution time analysis. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003, Proceedings*, pages 411–424, 2003. 2

[10] Open SystemC Initiative. Functional specification for systemc 2.0. *Outubro*, 2001. 4

[11] Erwan Jahier, Simplice Djoko-Djoko, Chaouki Maiza, and Eric Lafont. Environment-model based testing of control systems: Case studies. In *TACAS 2014*, Grenoble, France, April 2014. LNCS. 1

[12] Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Engineering functional requirements of reactive systems using synchronous languages. In *Int. Symp. on Industrial Embedded Systems*, Porto, Portugal, 2013. 1

[13] Erwan Jahier, Pascal Raymond, and Philippe Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer*, 8(6):517–530, 2006. 1, 3.2

[14] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. 6.1

[15] Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007. 2

[16] Raymond Pascal, Roux Yvan, and Jahier Erwan. Lutin: A language for specifying and executing reactive scenarios. *EURASIP-Journal on Embedded Systems*, 2008(1):753821, 2008. 1

[17] Peter P. Puschner and Roman Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 134–143, 1998. 2

[18] Tahiry Ratsiambahotra, Hugues Cassé, and Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. In *Performance Evaluation of Computer & Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, volume 41, pages 65–72. IEEE, 2009. 4

[19] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. Timing analysis enhancement for synchronous program. *Real-Time Systems*, 51(2):192–220, 2015. 8.1, 8.1

[20] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 430–444. Springer, 2008. 2

[21] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008. 1