Timing Analysis Enhancement for Synchronous Program*

Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux and Fabienne Carrier
Grenoble-Alpes University
Verimag, centre équation
2 avenue de Vignate, 38610 Gières
{firstname.lastname}@imag.fr

ABSTRACT

In real-time systems, an upper-bound on the execution time is mandatory to guarantee all timing constraints: a bound on the Worst-Case Execution Time (WCET). High-level synchronous approaches are usually used to design hard real-time systems and specifically critical ones. Timing analysis used for WCET estimates are based on the executable binary program. Thus, a large part of semantic information, known at the design level, is lost due to the compilation scheme (typically organized in two stages, from high-level model to C, and then binary code). In this paper, we aim at improving the estimated WCET by taking benefit from high-level information. We integrate an existing verification tool to check the feasibility of the worst-case path. Based on a realistic example, we show that there is a large possible improvement for a reasonable analysis time overhead.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Algorithms, Measurement, Verification

Keywords

WCET, Model-Based Design, Synchronous Languages, Model Checking, Traceability

1. INTRODUCTION

Hard real-time systems are generally built using model-based design. Particularly, control engineering systems are often generated from synchronous designs. In hard real-time systems any execution must fulfil timing constraints. To guarantee that these constraints are respected, a bound on the Worst-Case Execution Time is necessary.

Static timing analyses aim at estimating this upper-bound on the WCET. They are based on abstractions of the hardware and the software. They generally suffer the necessary over-estimation due to abstraction. The source of this over-estimation is twofold: the hardware model may generate over-estimation when joining abstract states, the semantics of the program may generate over-estimation due to the fact that the execution path corresponding to the estimated WCET may be infeasible. In this paper, we aim at improving static WCET analysis when the program under analysis has been generated from a synchronous model. We improve the WCET estimation by reducing the set of infeasible paths. We do not focus on the hardware analysis that we consider orthogonal.

WCET analyses are derived on the binary level. When programs are generated from high-level design, they are first translated into an intermediate language (usually C or ADA) then compiled into binary. Due to this two-step compilation, it may appear that most of semantic information is lost: the WCET estimation should be enhanced by considering this high-level semantic information. Two main issues must be solved to integrate these high-level semantic properties: (i) how to extract interesting semantic properties at the high-level, (ii) how to transfer information from one level to the lower next level (traceability).

This paper is structured as follows. First, we introduce the context: WCET analysis, synchronous model, and why synchronous model is a good candidate to enhance WCET estimation. Then, we show on a realistic example that the WCET estimation may be largely refined and we introduce our approach for programs generated from Lustre [5].

2. CONTEXT

2.1 WCET/Timing Analysis

In this paper we consider static timing analysis. Figure 1 shows the general timing analysis workflow that is used in a large part of WCET tools [15]. The analysis is based on the binary program. The Control Flow Graph (CFG) is first reconstructed from the binary. Then, a set of value analyses extracts memory addresses, loop bounds and simple infeasible paths. These analyses are mainly based on the binary or the C files (note that in case the C files are used some traceability analysis is necessary).

*This work is supported by the french research fundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
RTNS 2013 , October 16 - 18 2013, Sophia Antipolis, France
Copyright 2013 ACM 978-1-4503-2058-0/13/10 ...$15.00.
http://dx.doi.org/978-1-4503-2058-0/13/10.

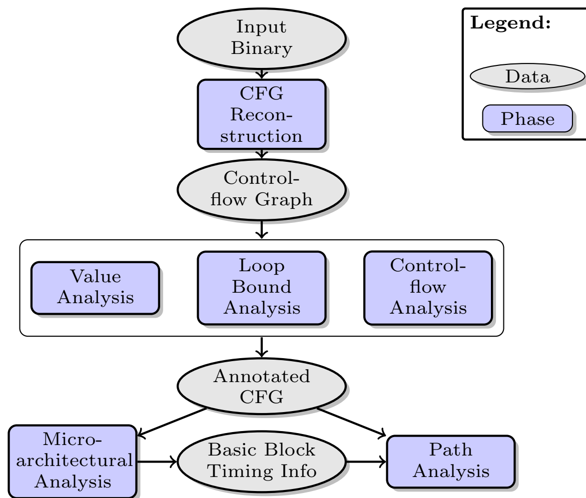


Figure 1: WCET analysis workflow

This semantic and address information is then helpful to proceed to the micro-architectural analysis. This analysis mainly estimates the execution time of basic blocks taking into account the whole architecture of the platform (pipeline, caches, buses,...). Note that, in this paper, we consider mono-processor platforms. The last step of the analysis uses the basic block execution time and semantic information to build the whole worst-case execution time.

In this paper we are contributing in this last step and the value/control-flow analysis. We extract semantic information from design-level (semantic analysis) and add them to the path analysis. In the rest of this section, we detail the most popular path analysis: the implicit path enumeration technique. More information about previous steps or different path analyses may be found in state of the art papers [15, 2].

Implicit Path Enumeration Technique. The implicit path enumeration technique is based on integer linear programming (referred as ILP in the sequel). The main idea is to associate to each block and each edge of the CFG, (1) an estimation of their local execution time, (2) a numerical variable denoting their number of executions (block) or traversal (edge). The total execution time is then defined as the sum of these variables, weighted by their local execution time.

The system of constraints is a set of structural constraints: two constraints per block, one for entry edges, one for exit edges. For instance for block 9 of Figure 7 (right part):

$$x_9 = edge_{7,9} + edge_{8,9} = edge_{9,21}$$

where x_i represents the number of executions of block i and $edge_{i,j}$ the number of executions of the edge from block i to block j. The estimated worst-case execution time is obtained by maximizing the following objective function that computes the execution time of the program:

$$\sum_{j \in \mathcal{B}} \sum_{i \in pred(j)} t_{i,j} * edge_{i,j}$$

where $\mathcal{B}$ is the set of all blocks in the CFG, $pred(j)$ gives the set of blocks in $\mathcal{B}$ that precede basic block j in the CFG, $t_{i,j}$ is the worst-case execution time of basic block j when block i is the previous executed block.

In this paper we use the tool OTAWA[1]. It gives us the CFG, all $t_{i,j}$ and the ILP set of constraints that describes the CFG and contains the objective function. We name ilp_{cfg} this initial set of constraints.

2.2 Synchronous programming

The synchronous paradigm was proposed in the early 80's with the aim of making easier the development of safety critical control systems. The main idea is to propose an idealized vision of time and concurrency that helps the programmer to focus on functional concerns (does the program compute right?), while abstracting away real-time concerns (does the system compute fast enough?). The goal of this section is to list the characteristics of synchronous languages that are of interest for timing analysis. For a more general presentation see [8, 5].

Semantically, a synchronous program is a reactive state machine: it reacts to its inputs by updating its internal state (internal variables) and producing the corresponding outputs. The sequence of reactions defines a global notion of (discrete) time, often called the basic clock of the program. Reactions are deterministic, which means that the semantics can be captured by *a transition function* $(s_{k+1}, o_k) = F(s_k, i_k)$, meaning that at the reaction number k, both the current outputs o_k and the next state s_{k+1} are completely determined by the current inputs i_k and state s_k. Moreover, the program must be well initialized: the initial state (s_0) is also completely determined.

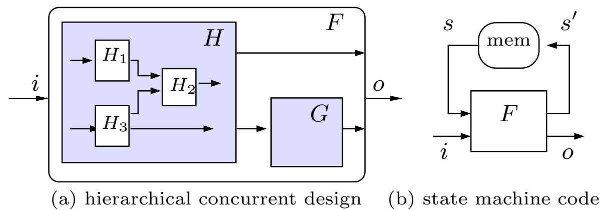
Practically, synchronous languages are providing features for designing complex programs in a concise and structured way. There are mainly two programming paradigms: in data-flow languages (e.g. Lustre, Scade[2], Signal [7]), programs are designed as networks of operators communicating through explicit wires; in control-flow languages, programs are designed as sets of automata communicating via special variables, generally called signals (e.g. Esterel [5], SynchCharts[3], Scade V6[2]). The style may vary, but the languages are all based on the same principles:

Synchronous concurrency: all the components of a program communicate and react simultaneously;

Causality checking: since all communications occur at the same "logical instant", a causality analysis is necessary to check whether the whole behavior is deterministic or not, in which case the program is rejected. This analysis is based on data-dependency analysis and may be more or less sophisticated depending on the language/compiler;

Compilation into sequential code: the basic compilation scheme for all synchronous languages consists in producing purely sequential code from the parallel source design; this generation relies on the causality analysis, which, if it succeeds, guarantees that there exists a computation order compatible with the data dependencies (this principle is often referred as *static scheduling*).

[1]http://www.otawa.fr
[2]http://www.esterel-technologies.com/products/scade-suite/
[3]http://www.i3s.unice.fr/ map/WEBSPORTS/SyncCharts/



(a) hierarchical concurrent design (b) state machine code

Figure 2: Synchronous compilation.

To summarize, the compilation of synchronous programs implements the semantics of the program as defined before: it identifies and generates the *memory* and the *transition function* of the program. Figure 2 illustrates this principle for a data-flow design: the hierarchic concurrent design (a) is compiled into a simple state machine code (b); according to the data dependencies (arrows), a possible sequential code for the function F is $H(); G()$, where $H() = H_1(); H_3(); H_2()$.

The synchronous compiler only produces the transition function, that is, the function that performs a single step of the logical clock. The design of the main program, responsible of iterating the steps, is left to the user. As a matter of fact, this loop strongly depends on architectural choices and on the underlying operating system. A typical choice consists in embedding the transition function into a periodic task activated on a real-time system clock.

Note that the principles presented in this section are valid for a large class of Model Based Design methods. As soon as a modeling language is equipped with an automatic code generator following the principles of Figure 2, it becomes a "de facto synchronous language". For instance, it is the case for the well-known Simulink/Stateflow[4] environment, which was originally designed for simulation purpose, and has been completed with code generators.

2.3 Timing analysis of Synchronous programs

The synchronous approach permits an orthogonal separation of concerns between functional design and timing analysis. The design and compilation method produces a code (transition function) which is *intrinsically real-time*, in the sense that it guarantees the existence of a WCET bound whatever be the actual execution platform. This property is achieved because the languages are *voluntary restricted* in order to forbid any source of unboundedness: no dynamic allocation, no recursion, no "while" loops. More precisely, the characteristics of a typical generated transition function are the following:

- the flow graph is mainly sequential, made of basic statements and nested conditionals (if, switch);
- synchronous languages allow to declare and manipulate statically bounded arrays, that are naturally implemented using statically bounded *for* loops;
- the code generation may be modular, in which case it contains function calls: the main transition function calls the transition functions of its sub-programs and so on; however this calling tree is bounded and statically known.

[4]http://www.mathworks.fr/products/stateflow/

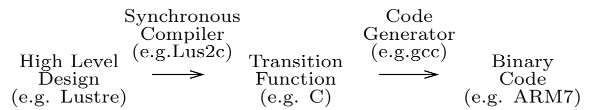


Figure 3: Typical 2-stage code generation in Model Based Design

Note that, besides these clean and appealing principles, synchronous languages also allow the user to import external code, in which case it becomes his/her responsibility to guarantee the validity of the real-time property.

The role of timing analysis is then to estimate an actual WCET for a particular platform. Synchronous generated code is particularly favorable for WCET estimation, since it is free of complex features (no heap, complex aliasing, loops, nor recursion). The goal of this paper is to try to go further, by exploiting not only *generic properties* of synchronous programs, but also *functional properties*. We start from the statement that static analysis of high-level synchronous programs has made important progress during the last decades: there exist checking tools (Model-checking, Abstract interpretation) able to discover or verify non-trivial functional properties, in particular *invariants* on the state-space of the program. This statement raises the following question: can such properties be exploited to enhance the timing analysis? Another statement is that such properties are hard or even impossible to discover at the binary level, and thus cannot be handled by existing timing analysis tools.

More precisely, the binary code is obtained via a two-stage compilation. Figure 3 illustrates this typical scheme, which is valid for all synchronous languages, and more generally for most Model-Based Design methods: high-level design is compiled into an "agnostic" general purpose language (C, most of the time), and then compiled for a specific binary platform. We give as example the languages and compilers that are actually used in this work (Lustre and its compiler, C and gcc, and ARM7 as the target processor).

WCET estimation is performed at the binary level, and enhancing the estimation mainly consists in rejecting (pruning) execution paths on the binary. This process raises several problems:

Control structure. There must exist conditional branches in the binary, otherwise there is nothing to prune. It supposes that the synchronous compiler actually generates a control structure, that will later be compiled into conditional branches. This fact strongly depends on the high-level language and its compiler. Control flow languages (Esterel, Scade 6) provide explicit control structures that are likely to be mapped into C, and then binary control structures. In data-flow languages, the control structure is implicit: conditional computation is expressed in terms of clock-enable conditions, similarly to what happens in circuit design. The compiler must generate conditional statements, but the whole structure is likely to be less detailed than the one produced from a high level control-flow language. In this paper, we focus on this less favorable case, by considering examples written in Lustre.

Non trivial functional properties. At the high level, we must be able to discover and/or check properties that are hard to find at the binary level. For this purpose, state invariants are clearly good candidates: they are related to the fact that the program state is initialized with a known value, and that it is only modified by successive calls to the transition function. All this information is indeed not available for the WCET analyser.

Traceability. Supposing that the compilation produces a suitable control structure, and that we can find interesting functional properties, there remains the technical problem of relating the properties with the binary code branches. This traceability problem is made more complex because of the two stage compilation: if we can expect a relatively simple traceability between Lustre and C, this is not the case between C and the binary code, in particular because of the code optimizations that may be rather intrusive.

All these problems and the proposed solutions are detailed in section 4.

3. RELATED WORK

As far as the authors know there is no previous work specifically targeting the enhancement of timing analysis for synchronous data-flow languages (SCADE/Lustre).

In [6] the AiT WCET tool[5] is integrated in the SCADE suite for analyzing the binary programs generated from SCADE. This work pays attention to traceability but does not check the feasibility of estimated worst-case paths according to the high-level semantics.

Other related work focus specifically on the Esterel language. As explained in Section 2.3 and in [13], Esterel is a control-flow language; the timing analysis is easier than for data-flow language due to the fact that the mapping of high-level control-flow graph and the C level is almost one-to-one. Thus, the analysis may be based on the high-level CFG through some graph traversal analysis and pattern matching [13, 10, 4]. The existing analyses of Matlab/simulink programs are also mainly based on the "control-flow" features of this language with some annotations about the execution modes [14, 11].

The analysis of synchronous programs may gain some semantic information by considering in the context of a step the state of the memory at the end of the previous step [9]. This work is orthogonal to our approach. As a future work, we will extend our approach with this "memory" consideration.

Some related works use model-checking to analyse WCET [15]. Among them: in [1] PRET-C concurrent programs are analysed; in [3] dynamic analysis is combined to model-checking to get timing and path information.

Finally, none of the cited work relies on optimization from the C level to binary level. They suppose that there is no optimization and limit their path analysis at the C level while we start from binary level.

4. PROOF-OF-CONCEPT ON A REALISTIC EXAMPLE

This section details the methods developed for this study, through the treatment of a typical example. We do not claim

[5]http://www.absint.com/

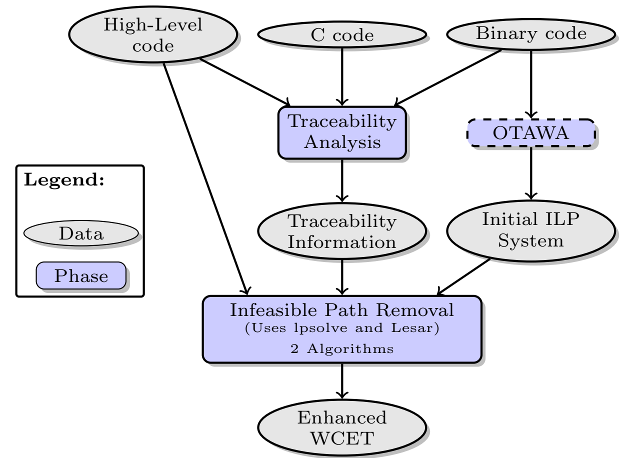


Figure 4: Proof-of-concept Workflow

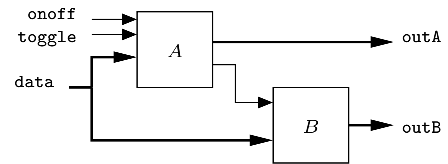


Figure 5: Overall organization of a typical control system.

to propose a complete "turnkey" solution, but rather a proof of concept that illustrates how, and how much, WCET can be enhanced by exploiting high-level functional properties.

Figure 4 presents the general workflow of the experiment presented in this section. From the source high level code, the generated C code, and the corresponding binary code, the *Traceability Analysis* tool extracts information relating high-level expressions to binary branches (§ 4.2). In the meanwhile, the external tool OTAWA analyses the binary code and builds the initial WCET problem (expressed as an Integer Linear Programming optimization System). Using the high-level code and the traceability information, the *Infeasible Path Removal* tool tries to refine the initial ILP system in order to obtain an enhanced WCET estimation. This tools provides several algorithms/heuristics and uses the external tools `lp_solve` to solve ILP systems and `Lesar` to model-check properties of the high-level code (§ 4.6).

4.1 Example

This example is a simplified version of a typical control engineering application. The overall organisation is shown in Figure 5. At higher level, a program is designed as a hierarchy of concurrent sub-programs dedicated to a particular "task"; the example has two concurrent tasks, A and B, performing treatments on the same input data, according to input control commands (`onoff`, `toggle`). Tasks are in general not independent: in the example, B depends on a value produced by A. One role of the high-level compiler is to correctly schedule the code according to these dependencies (§2.2).

Figure 6 shows the core of sub-program A. This program

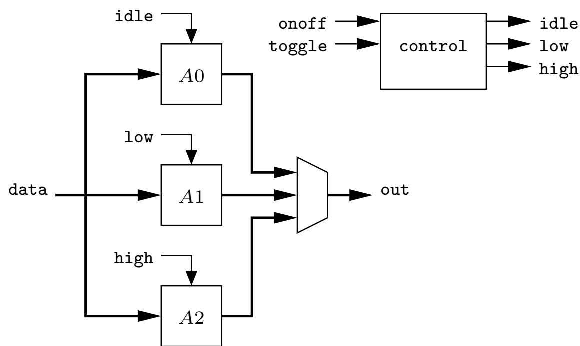


Figure 6: Submodule A behaves according to 3 operating modes.

performs several treatments on its input `data` according to several *operating modes*. The data part is kept abstracted, but we gave meaningful names to the control part in order to make their role clearer. The small arrow on the top of a block behaves as a clock-enable: the block is activated (i.e., executed) if and only if the clock is true. In the example, `idle` (resp. `low` and `high`), enables the computation of $A0$ (resp. $A1$ and $A2$). The clocks are themselves computed by a `control` logic, according to the input commands `onoff` and `toggle`. We don't detail the code corresponding to `control`, but roughly, the command `onoff` switches between `idle` or not, and, when not `idle`, `toggle` switches between `low` and `high` mode. What is really important for the sequel, is that the controller satisfies the following high-level requirement: if the inputs `onoff` and `toggle` are assumed exclusive, then the program guarantees that `idle`, `low` and `high` are also (pairwise) exclusive.

Sub-program B is similar to A except that it has only two operating modes: when `nom` (nominal mode) it computes the function $B0$, and when `degr` (degraded mode) it computes $B1$. The control logic is also simple: the mode switches from nominal to degraded (and conversely), whenever the input command `onoff` is activated. This last point is important since it introduces a high-level property relating the modes of A and the modes of B: B must be in mode `degr` whenever A is not in mode `idle`.

Finally, this example, while relatively simple, is a good candidate for experimenting since it satisfies high-level properties that are likely to enhance the WCET estimation:

- whatever be the details of the compilation (from Lustre to C, and C to binary), high-level control variables (clock-enables) will certainly be implemented by means of conditional statements,
- high-level relations between clock-enables (exclusivity) are likely to make some execution paths infeasible, and then, the WCET estimation should be enhanced.

For this experiment, the example has been developed in Lustre, in order to use the associated tool chain, mainly the Lustre to C compiler `lus2c`[6] and the Lustre model-checker `Lesar`[12].

[6]http://www-verimag.imag.fr/The-Lustre-Toolbox.html

Before considering the whole program, we analyse the WCET of the different modes, in order to foresee which combination of modes is likely to be the most costly. It appears that the cost of the combinations $(A2 + B1)$ and $(A0 + B0)$ are both very close, and much higher than any other possible combination. Thus, the worst case is expected for one of these cases.

The first problem (relating high-level variables to binary branches) is addressed as the *traceability* problem in the following. When binary branches are related to high-level variables, the next step consists in developing an automated method for enhancing the WCET estimation (§ 4.6).

4.2 Traceability

For this experiment, we use a state of the art C compiler, `arm-elf-gcc 4.4.2`, a widely used cross compiler for the ARM7 platform.

First of all, we have to define precisely what the traceability problem is: for a given edge in the binary control flow graph (e.g., basic block i to basic block j, noted $edge_{i,j}$), find, if possible, a Boolean Lustre expression (e.g. `toggle and not onoff`), such that the binary branch is taken iff the Lustre expression is true. We have developed a prototype tool to solve this problem, for which we only present here he main principles. Because of the two-stages compilation, the problem requires to trace information at two levels: from Lustre to C, and then C to binary.

From Lustre to C. This problem is easily solved since we control the development of the `lus2c` compiler. The compiler has been adapted in such a way that all `if` statements in the generated C code get the form `if(Lk)` where `Lk` is a local C variable; moreover, a pragma is generated in order to associate this variable to the source expression it comes from. We note $\mathcal{E}(Lk)$ this Lustre expression. The semantics of such a pragma is: the value of the C variable at this particular point of the C program is exactly the value of the Lustre expression. For a sake of simplicity, we no longer make a difference between the C variable and the corresponding expression: we talk about the Lustre condition controlling the C statement.

From C to binary. The problem, while mainly technical, can hardly been addressed here in a completely satisfactory way: patching the gcc compiler to exactly add the information we need would require an amount of work which is out of the scope of this academic study. We have then chosen to rely on the existing debugging features to relate binary code to the C code. Indeed, code optimization may dramatically obfuscate the C control structure within the binary. The retained solution is then not complete but safe: whenever a binary choice can be related, via the debugging information, to a corresponding `if` statement in the C code, we keep this information. Otherwise the choice is simply ignored. While naive, this solution resists to relatively intrusive optimizations performed by gcc on the control structure.

4.2.1 Traceability without optimization

Without optimization (`-O0` option of gcc), as expected, the binary control flow graph strictly maps the C control flow graph.

Figure 7 shows the C CFG (left) and the binary CFG (right) of the example. Both graphs have been simplified

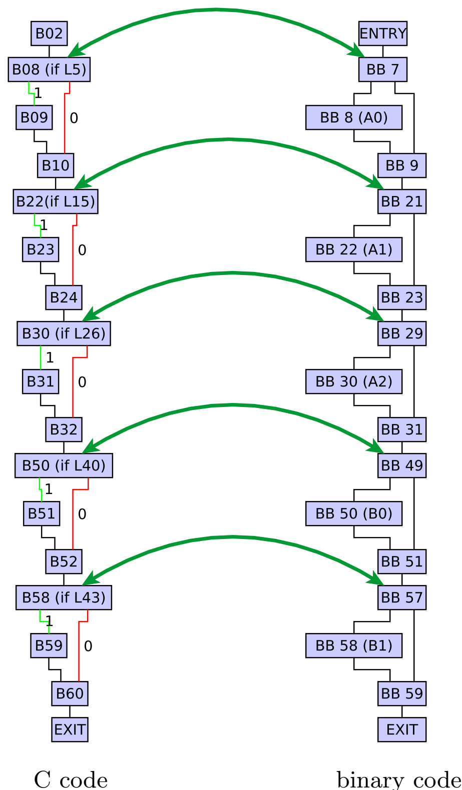


Figure 7: CFG traceability with -O0

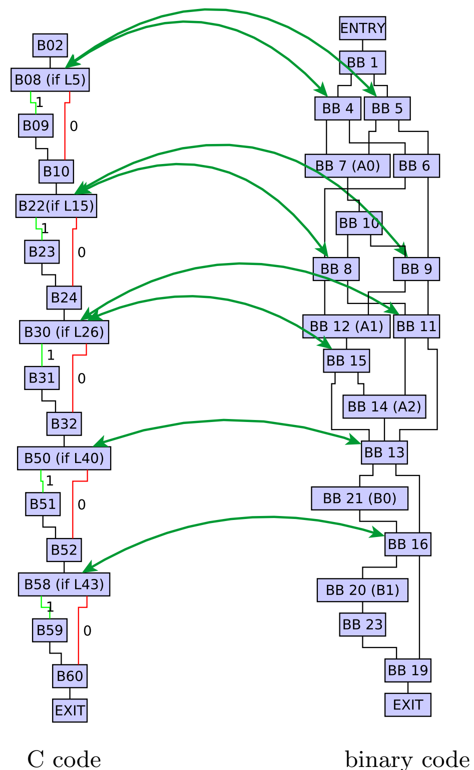


Figure 8: CFG traceability with -O2

in order to outline parts of interest (the actual graphs have more than 70 nodes). Only the branches concerning the calls of interest ($A0$, $A1$, etc) are depicted. Unsurprisingly, a one-to-one correspondence is automatically found by our tool, based on the gcc debugging information.

4.2.2 *Traceability with optimization*

Using optimizations raises problems in critical domains submitted to certification process. For instance in avionics, the highest safety levels of the DO-178B document[7] specify that traceability is mandatory from requirements to all source or executable code. Discussing whether some optimization is reasonable in critical domains is out of scope of this paper: we only aim at experimenting how optimizations may affect traceability, and thus, be an obstacle for enhancing WCET estimation.

Figure 8 shows the C CFG and the binary CFG obtained with `gcc -O2`. The most remarkable effect of the optimization concerns the first part (computation of A): the source sequence of 3 conditionals is implemented by a structure of interleaved jumps. In this graph, some branches of binary code get no debugging information at all and thus, cannot be related to the source code (BB1 and BB10). Conversely some source tests have been duplicated in the target code (e.g. BB4 and BB5 are attached to the same source test). The goal of this kind of transformation is to give priority (in average) to program-counter increment (sequence) over costly program-counter jump. Even relatively intrusive, this transformation does not affect traceability: the binary choice is related to its source C condition, but, indeed, several branches may refer to the same source.

[7]http://www.rtca.org/store_list.asp

4.3 From paths to predicates

At this point, we suppose that traceability analysis has been achieved. The result is a partial function from CFG edges to C literals (either `Lk` or `not Lk`), and then, to the corresponding Lustre conditions (either $\mathcal{E}(\texttt{Lk})$ or $\neg\mathcal{E}(\texttt{Lk})$).

We note $cond_{i,j}$ the Lustre condition corresponding to $edge_{i,j}$ (if it exists). Note that if $edge_{i,j}$ and $edge_{i,k}$ are the two edges of a binary choice, if $cond_{i,j}$ exists, then $cond_{i,k}$ exists too, and $cond_{i,j} = \neg cond_{i,k}$. Here are, for instance, some of the 16 traceability "facts" found on the optimized code of the example (see Figure 8):

$$cond_{4,7} = cond_{5,7} = \mathcal{E}(\texttt{L5})$$

$$cond_{4,6} = cond_{5,6} = \neg\mathcal{E}(\texttt{L5})$$

$$\cdots$$

$$cond_{16,20} = \mathcal{E}(\texttt{L43})$$

$$cond_{16,19} = \neg\mathcal{E}(\texttt{L43})$$

In the sequel, for the sake of simplicity, we extend the traceability function by assigning the condition "true" to any edge that is not related to a Lustre condition. For instance, in the example: $cond_{entry,1} = cond_{1,4} = cond_{1,5} = \cdots =$ *true*.

4.4 Checking paths feasibility

A path (or more generally a set of paths) in the CFG can be described by a set of edges. Consider the binary CFG in the case `-O2` (Fig. 8); the set of edges

$\{edge_{1,4}, edge_{4,7}, edge_{8,12}\}$ corresponds to the set of paths that passes by those 3 edges. In particular, for this set of paths, basic blocks 7 ($A0$) and 12 ($A1$) are both executed. For such a set $\{edge_{i_k,j_k}\}$, one can build a Lustre logical expression expressing the feasibility of the paths: $\bigwedge_k cond_{i_k,j_k}$.

The idea is then to check whether this Boolean expression is satisfiable according to the knowledge we have on the program. In some cases the answer is trivial: when the expression syntactically contains both a condition `Lk` and its negation ¬`Lk`, the corresponding edges are *trivially exclusive*. Some of these trivial exclusions are already taken into account as structural constraints: this is obviously the case for conditions guarding the two outcoming branches of a same node (e.g. $edge_{i,j}$ and $edge_{i,k}$). However, this is not always the case: in the C code, the same condition may be tested several times along the control graph, in which case the logical exclusion is not "hard-coded" in the structure.

When infeasibility is not trivial, it is necessary to use a decision tool. More precisely, we build the predicate $infeasible = \neg(\bigwedge_k cond_{i_k,j_k})$, and ask the model-checker (Lesar) to prove that *infeasible* is an invariant of the program. The model-checker (just like any another automatic desision tool) is partial; it may answer:

"yes": in which case we know that *infeasible* remains true for any execution of the program, and, thus the corresponding paths can be pruned out.

"inconclusive": in which case *infeasible* may or may not be an invariant.

In the example, the model-checker answers "yes" for the infeasibility predicate $\neg(cond_{4,7} \wedge cond_{8,12})$. As expected from the program specification, it means that blocks 7 (call of $A0$) and 12 (call of $A1$) are never both executed.

One of the main argument for the proposed method is that such property is almost impossible to discover at the binary or C level: this is a relatively complex consequence of both assumptions on inputs and dynamics of the underlying state machine:

- inputs `onoff` and `toggle` are assumed exclusive,
- `idle` and `low` are initially exclusive (resp. true and false),
- whatever be an execution of the program satisfying the assumption, the computation of `idle` and `low` ensures that they remain exclusive in any reachable state.

The model-checker also covers simpler *static* logical properties, for instance:

$$\neg((X \Rightarrow Y) \wedge X \wedge \neg Y),$$

and even simple numerical properties[8]:

$$\neg((X > 5) \wedge (3X \leq 14))$$

4.5 From infeasibility to linear constraints

In this case study, we limit the use of the discovered properties to the pruning of infeasible paths in the last stage of the WCET estimation, that is, at the ILP level. For ILP solving, we use the tool `lp_solve`[9].

Suppose that we have proved $\neg(\bigwedge_k cond_{i_k,j_k})$; it means that there exists no path passing by all the corresponding edges $\{edge_{i_k,j_k}\}$.

In the ILP system, each edge is associated to a numerical variable representing the number of times the edge is traversed during an execution; the same notation $edge_{i_k,j_k}$ is used for these numerical variables (context avoids misleading). This section presents how to translate the logical expression into a numerical constraint.

In the general case (programs with loops), translating the exclusivity of a set of edges $\{edge_{i_k,j_k}\}$ requires an extra information: *a structural max of the set*. Intuitively, a structural max is an upper bound of the number of executions that pass through at least one of the edges. Given such structural max μ, the exclusivity property can be translated into a numerical constraint:

$$\textstyle\sum_{k=1}^{n} edge_{i_k,j_k} < n \times \mu$$

The smallest is the bound, the most precise is the constraint. In the case of loop-free programs (which is the case for our example and, more generally, for any transition function not using arrays), 1 is a structural max for any set of edges. For a set of n edges, the ILP constraint is:

$$\textstyle\sum_{k=1}^{n} edge_{i_k,j_k} < n$$

For instance, the proven invariant $\neg(cond_{4,7} \wedge cond_{8,12})$ is translated into:

$$edge_{4,7} + edge_{8,12} < 2$$

4.6 Algorithms and strategies

We have seen so far: how to use a model-checker to check infeasibility properties, and how to translate such properties into numerical constraints for the ILP solver. The problem is now to define a complete algorithm for the WCET estimation, that decides which properties to check and when to check them.

In the sequel, we use the following notations:

- "$BinaryAnalysis(bprg) \rightarrow ilp_{cfg}$" is the abstraction for the main OTAWA procedure that analyses a binary program and returns a whole initial integer linear program (i.e., linear constraints + objective function), a linear constraint is noted *ilc*,
- $CheckInv(lprg, lexp) \rightarrow yes/no$ represents the call of the Lesar model-checker; it takes a Lustre program and a Lustre predicate, and returns yes if the predicate is a proven invariant of the program, no otherwise,
- $LPSolve(ilp) \rightarrow wcep$ represents the call of the ILP solver; it returns the worst-case execution path,
- $ConditionOfPath(wcep) \rightarrow lexp$ is the procedure that traces back a binary path to the corresponding conjunction of Lustre literals, as explained in Section 4.4,
- $ConstraintOfPath(wcep) \rightarrow ilc$ is the procedure that takes a binary path and produces the integer linear constraint that will be used to state the infeasibility of the path, as explained in 4.5.

4.6.1 Removing trivial exclusions

A first step, which is almost free once the traceability step has been done, consists in taking into account trivial exclusions of the form `Ln` and ¬`Ln`. As explained in Section 4.4, some of these trivial exclusions are also structural, and thus

[8]More precisely, Lesar is equipped with a numerical solver that handles Linear Algebra Theory.

[9]http://lpsolve.sourceforge.net/

the constraint is already taken into account in the ilp_{cfg} system. However other trivial exclusions are not structural: they are due to the Lustre compilation that may open and close the same test several times along the execution paths.

For all pairs of edges with opposite conditions, that are not structurally exclusive, we generate the ILP constraint $edge_{i,j} + edge_{k,l} < 2$. The effect of this first enhancement strongly depends on the optimization level:

- In `-O0` case, traceability identifies 9 conditions: the 5 depicted on the simplified CFG (Fig. 7) plus 4 more. Some conditions are tested several times along the execution path. For instance, one of the extra condition (named `M7`), and that intuitively controls some initializations, is tested not less than 7 times. Other conditions are tested 4 times (`L5`, `L15` etc). Finally, 102 trivial (but not structural) exclusions are automatically discovered and translated into ILP constraints. The WCET estimation is slightly enhanced: 4726 to 4701 cycles. The enhancement is not impressive, but the number of possible paths is once and for all dramatically reduced: by considering the details of the branches, we know that these constraints are dividing by $2^{17}(131.072)$ the number of feasible paths.
- In `-O2`, traceability only identifies 6 conditions, the ones outlined on Figure 8, plus the `M7` presented above. The fact that 3 conditions from the C code have disappeared in the binary code is due to an optimization related to the target processor capabilities: ARM7 provides a conditional version for most of its basic instructions, and the gcc compiler can then replace simple conditional statements with (even simpler) conditional instructions. Since the resulting binary CFG is much simpler than the one obtained with `-O0`, the result is less impressive: no trivial exclusion is discovered which is not already a structural exclusion. This first step gives a non-enhanced WCET estimation (762 cycles, which is OTAWA initial estimation).

4.6.2 Iterative refutation algorithm

First, we experiment a *refinement algorithm* which iterates *LPSolve* and Lesar calls to obtain a candidate worst case and try to refute it. The pseudo-code of the algorithm is the following:

Algorithm 4.1: *RefineWCET*$(lprg, bprg)$

$ilp \leftarrow BinaryAnalysis(bprg)$
while *true*
do
  $wcep \leftarrow LPSolve(ilp)$
  $lexp \leftarrow ConditionOfPath(wcep)$
  $infeasible \leftarrow CheckInv(lprg, \neg lexp)$
  if *infeasible*
    then $ilp \leftarrow ilp \cup ConstraintOfPath(wcep)$
    else return (ilp)

The result of this algorithm is optimal modulo the decision procedure: it converges to a worst-case path which is actually *feasible* according to the decision procedure.

The obvious drawback of the method is the number of necessary iterations, which can grow in a combinatorial way. Our example clearly illustrates this problem. Let us consider the `-O0`, and focus only on the interesting branches (binary cfg in Figure 7). Unsurprisingly, the initial worst-case path found by *LPSolve* corresponds to a case where all modes are executed:

$$edge_{7,8}, edge_{21,22}, edge_{29,30}, edge_{49,50}, edge_{57,58}$$

This path is refuted by the model checker, and a new numerical constraint is added to the ILP problem:

$$edge_{7,8} + edge_{21,22} + edge_{29,30} + edge_{49,50} + edge_{57,58} < 5$$

In return, the ILP finds another worst case where $edge_{7,8}$ is replaced by $edge_{7,9}$, in other terms, all but the least costly mode are executed ($A0$). This path is refuted and a new constraint is added:

$$edge_{7,9} + edge_{21,22} + edge_{29,30} + edge_{49,50} + edge_{57,58} < 5$$

which leads to another false worst case where all modes but $A1$ are executed (the second in the cost order), and so on. To summarize: the algorithm first enumerates all the cases where all modes but one are executed, then where all modes but two are executed, and finally all but three. It finally reaches the real worst case where exactly 2 modes (actually $A2$ and $B1$) are executed.

In the example, the algorithm behaves slightly differently depending on the code optimization level:

- With `-O0`, the algorithm converges in 298 steps, from a first estimation of 4701 (after the trivial exclusion removal §4.6.1) to a final optimal estimation of 2375. As expected, the worst case path is the one where modes $A2$ and $B1$ are executed.
- With `-O2`, the algorithm converges after 114 steps, from an estimation of 762 to 459 cycles. As expected, the worst case path is the one where modes $A0$ and $B0$ are executed.

This experiment outlines the combinational cost of the method. This problem arises because the iterative constraints are not precise enough: for instance, a constraint states that at most 4 of 5 edges can be taken, where the "right" information is that they are all *pairwise* exclusive.

4.6.3 Searching properties before WCET estimation

In this section, rather than refuting an already obtained worst-case path, we study the possibility of inferring, *a priori*, a set of high-level properties that are likely to help the forthcoming search of worst-case path. More precisely, we consider that a set of relevant high-level conditions has been selected. This selection may be greedy (all the identified conditions), or more sophisticated (conditions controlling "big" pieces of code).

In our simple example, both heuristics lead to select the same set of interesting conditions: the ones controling the modes {`L5`, `L15`, `L26`, `L40`, `L43`} (Fig. 7 and 8), plus some others depending on the code optimization (4 more in case `-O0`, 1 more in case `-O2`).

Useful information on these variables are *disjunctive relations* (or clauses): if one can prove that a disjunction is an invariant (e.g. ¬`L15` ∨ ¬`L26` ∨ ¬`L43`), its negation (conjunction) is proven impossible and the corresponding paths can be pruned.

A virtual complete method. Using a model checker, it is theoretically possible to find an "optimal set of invariant clauses" over a set of conditions L_k (optimality is relative to the decision procedure capabilities). The sketch of the algorithm is the following:

- let *lprg* be the considered Lustre program, and s be its state variables; a model-checker can compute a formula $Reach(s)$ denoting a superset of the reachable states of the program; in some cases (e.g. finite-state programs) the formula is exact, but in general (e.g. numerical values) the formula is a strict over-approximation;
- let L_k be the conditions of interest, and $e_k(s) = \mathcal{E}(L_k)$ the corresponding Lustre expressions; these expressions are functions of the program state variables;
- consider the following formula over the state variables and the conditions:

  $$Reach(s) \bigwedge_{k \in K} (L_k = e_k(s))$$

  Quantifying existentially the state variables projects the formula on the L_k variables only, giving the set of all possible configurations for the L_k variables:

  $$\Phi(L_k) = \exists s \ \cdot \ Reach(s) \bigwedge_{k \in K} (L_k = e_k(s))$$

- formula $\Phi(L_k)$ can be written in conjunctive normal form (i.e conjunction of clauses):

  $$\phi(L_k) = \bigwedge_{i \in I} D_i(L_k)$$

  involving a *minimal* number of *minimal clauses* (this is the dual problem of finding a minimal prime implicants cover for disjunctive normal forms);
- each minimal clause can be translated into a minimal infeasibility constraint as explained in Section 4.5.

This algorithm combines several well-known intractable problems (reachability analysis, minimal clause cover) and remains largely virtual.

A pragmatic pairwise approach. We now focus on non-complete methods that may/may not find relevant relations. From a pragmatic point of view, two-variable clauses (i.e., pairwise disjunctive relations) are good candidates: the number of pairs remains relatively reasonable (quadratic), and their analysis is likely to remain reasonable (since they only involve a small portion of the full program).

We detail the algorithm on the example, in the case `-O2` (cf. Fig. 8). All the conditions identified by the traceability process are selected: the 5 emphasized in Fig. 8 (`L5`, `L15`, `L26`, `L40`, `L43`), plus an extra one, `M7`. Intuitively, this variable controls initializations; it has a strong influence on functionality, but not on the WCET. We build and check all the possible pairwise disjunctive relations between these 6 conditions, that is, $4 \times (6 \times 5)/2 = 60$ predicates. The proof succeeds for 14 relations, that reflect most of the properties we know from the program specification: exclusion between $B0$ and $B1$, pairwise exclusion for $A0$, $A1$ and $A2$, exclusion between $B0$ and either $A1$ and $A2$.

One property of the program is not reflected: the fact that at least one of the modes $A0$, $A1$, $A2$ is executed. This property is induced by a 3-variable disjunction, and thus, cannot be discovered by the method. However, this information is useless, since the worst case certainly not occurs when none of the modes are executed.

Another exclusion is discovered: $\neg$`M7` $\vee$ $\neg$`L26`. Intuitively, this property means that mode $A2$ cannot be executed at the very first reaction. This is an unexpected consequence of the specifications[10], which may have some influence on the worst-case path.

Finally, the 14 properties are translated into 114 ILP constraints (remember that the same condition may control several branches). With this additional constraints, `lp_solve` directly gives the same worst-case path and time obtained with the iterative algorithm. To summarize, this somehow brute-force method gives the same result with only 60 relatively cheap calls to Lesar, and a single call to *LPSolve*, where the *a priori* smarter refutation method requires 114 relatively costly calls to the model-checker and as many calls to *LPSolve*.

In case `-O0`, 9 atomic conditions are identified, leading to 144 binary disjunctions to check; 21 are proven invariant by Lesar, and translated into 396 ILP constraints. The final estimation is 2376, which misses the optimal one for 1 cycle. The path obtained is still not feasible: this is the drawback of the method which only consider pairwise properties and may miss more complex relations. However, if and when optimality is the goal, the algorithm can be completed with one (or more) steps of the iterative method.

4.7 Results and discussion

Table 1 gives some quantitative results on the experiment. Each line corresponds to an algorithm (complete iterative method, or pairwise exclusion method) and an optimization level of gcc.

The first 3 columns detail the WCET estimation (in cpu cycles): the initial estimation is performed without any additional information (initial OTAWA WCET), the first (enhanced) one is obtained by exploiting the trivial (syntactic) exclusions (§ 4.6.1), and the final estimation is obtained by exploiting all the information, including those coming from Lesar. For the 3 main tools involved in the method, Lesar, lp_solve and OTAWA, we give the number of necessary calls and their cost (total computation time). Note that OTAWA is called once in any case (not given in the table), and that the number of lp_solve calls is always the number of Lesar calls plus 2 (initial and first estimations). A column describing the complexity of ILP constraints is also given in order to better understand the influence of the methods on the ILP solver. For each case, we distinguish 3 sets of constraints, for which we give both their number and their arity (maximal number of variable appearing in a constraint): structural constraints describe the binary CFG, they are produced once for all by OTAWA and only depend on the optimization level. Trivial constraints are those that reflect simple exclusions and also depend only on the optimization level. At last, invariant constraints are those that come from the invariant properties checked by Lesar.

We can reasonably conclude, by comparing the cases `-O2` and `-O0`, that the iterative algorithm is unlikely to scale up for bigger codes. From the `-O2` code to the `-O0` code, the size grows roughly 3 times; in the same time, both the number of iterations and the total cost of Lesar seem to

[10] Since we assume that inputs are exclusives, it takes at least two reactions to reach this particular mode.

Algo/optim	Wcet estimation			Lesar		ILP Constraints			LPSolve		OTAWA	Total
	initial	first	final	calls	cost	struct.	triv.	inv.	calls	cost	cost	cost
iter./-O0	4726	4701	2375	298	2.8s	178 (2)	102 (2)	298 (30)	300	90s	64s	163s
iter/-O2	762	762	459	114	0.7s	60 (3)	6 (2)	114 (6)	116	1.5s	1s	4.5s
pair./-O0	4726	4701	2376	144	1.4s	178 (2)	102 (2)	396 (2)	3	0.1s	64s	67s
pair./-O2	762	762	459	60	0.6s	60 (3)	6 (2)	47 (2)	3	0.1s	1s	2.3s

Table 1: Experiment: quantitative results

grow linearly (factor between 2 and 3), but the total cost of LPSolve dramatically explode (multiplied by 60). More than the increasing number of constraints, the increasing of their complexity explains this explosion (298 constraints, each involving 30 variables).

On the contrary, the pairwise exclusion method behaves in a very promising way: when considering only the overhead due to our method, and not the cost of OTAWA, the cost seems to grow linearly.

5. CONCLUSION

In this paper, we introduced a method to improve timing analysis of programs generated from high-level synchronous design. The main idea is to take benefit from semantic information that are known at the design level and may be lost by the compilation steps. For that purpose, we use existing verification tools that work on Lustre programs to check the feasibility of paths. Furthermore, our approach may work on optimised code (from C to binary). We introduced the approach on a realistic example and showed that there is a huge possible improvement, with a reasonable overhead compared to the (unavoidable) cost of the binary code and architecture analyses.

As future work, we would like to extend our approach to other languages with a reacher set of high-level constructs (e.g. Scade V6), and to other code generators (e.g., Compcert[11]). The long term aim is to study the possibility of a complete WCET-aware compilation chain.

6. REFERENCES

[1] S. Andalam, P. Roop, and A. Girault. Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs. In *DATE*, 2011.

[2] M. Asavoae, C. Maiza, and P. Raymond. Program semantics in model-based wcet analysis: A state of the art perspective. In *WCET*, pages 31–40, 2013.

[3] J.-L. Béchennec and F. Cassez. Computation of wcet using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011.

[4] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *ENTCS*, 203(4):65–79, June 2008.

[5] P. Caspi, P. Raymond, and S. Tripakis. Synchronous programming. In I. Lee, J. Y.-T. Leung, and S. H. Son, editors, *Handbook of Real-Time amd Embedded Systems*, chapter 14. Chapman and Hall/CRC, 2007.

[6] C. Ferdinand, R. Heckmann, T. L. Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *ERTS2*, 2008.

[7] T. Gauthier, P. L. Guernic, and L. Besnard. Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture*. LNCS 274, Springer Verlag, 1987.

[8] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[9] L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of esterel programs. In *DAC*, pages 870–873, 2009.

[10] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of esterel specifications. In *CODES-ISSS*, 2008.

[11] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *ECRTS*, 2002.

[12] P. Raymond. Synchronous program verification with lustre/lesar. In S. Mertz and N. Navet, editors, *Modeling and Verification of Real-Time Systems*, chapter 6. ISTE/Wiley, 2008.

[13] T. Ringler. Static worst-case execution time analysis of synchronous programs. In *Ada-Europe*, pages 56–68, 2000.

[14] L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for Matlab Simulink/Stateflow. In *ACES-MB*, 2009.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.

[11] http://compcert.inria.fr