Timing Analysis Enhancement for Synchronous Program
Extended Abstract

Pascal Raymond, Claire Maiza,
Catherine Parent-Vigouroux,
Fabienne Carrier, and Mihail Asavoae

Grenoble-Alpes University
Verimag, Centre Équation
2 avenue de Vignate, 38610 Gières, France
`firstname.lastname@imag.fr`

Abstract. Real-time critical systems can be considered as correct if they compute both *right* and *fast enough*. Functionality aspects (computing right) can be addressed using high-level design methods, such as the synchronous approach that provides languages, compilers and verification tools. Real-time aspects (computing fast enough) can be addressed with static timing analysis, that aims at discovering safe bounds on the Worst-Case Execution Time (WCET) of the binary code. In this paper, we aim at improving the estimated WCET when the analyzed binary code comes from a high-level synchronous design. The key idea is that some high-level functional properties may imply that some execution paths of the binary code are actually infeasible, and thus, can be removed from the worst-case candidates. In order to automatize the method, we show (1) how to trace semantic information between the high-level design and the executable code, (2) how to use a model-checker to prove infeasibility of some execution paths, and (3) how to integrate such infeasibility information into an existing timing analysis framework. Based on a realistic example, we show that there is a large possible improvement for a reasonable computation time overhead.

1 Context: timing analysis and Synchronous Programming

1.1 WCET/Timing Analysis

We consider here state-of-the-art static Timing Analysis techniques ([1]), whose aim is to find a safe (and hopefully accurate) upper bound on the Worst-Case Execution Time of a function/procedure. WCET estimation tools are classically organized into several stages:

– Micro-architectural analysis takes the binary code as input, together with the model of the architecture. It builds the Control Flow graph of the program,

made of basic blocs of (purely sequential) instructions, connected by edges representing the control passing. At each vertex and/or edge is assigned a local worst-case execution time (aka a weight), expressed in cpu cycles.
– Flow analysis extracts information about execution paths in the CFG (dead code, infeasible path [2]). This stage must at least be able to bound the length of the executions, otherwise the weight will trivially be infinite (loop bounds).
– Implicit Path Enumeration Technique is the last stage, where the actual worst case path and time is computed [3]. This technique consists in encoding the problem into an Integer Linear Programming optimization problem: a numerical variable is assigned to each block/edge, whose value is the number of time it is executed/traversed during an execution. The ILP problem is made of structural constraints, directly deduced from the structure of the graph, and semantics constraints, that (at least) bound the execution of the loops. The objective function is to maximize the sum of the counter variables weighted by their local WCET.

1.2 Synchronous Programming

Synchronous Programming approach provides high-level languages compilers and validation tools for the design of safe, deterministic reactive systems [4]. There exist several programming styles (mainly data-flow or control-flow oriented), but the principles are always the same:

– The user designs its application with an idealized vision of time and concurrency: a logical discrete clock exists and each change or communication takes place at some instant of this discrete clock.
– A compiler takes this synchronous concurrent design and produces a *step procedure* that implements a single reaction of the whole system, consisting on an interleaving of the concurrent tasks (static scheduling).
– The languages are voluntarily limited in such a way that the code execution time is intrinsically bounded (no recursion, no dynamic memory allocation, only trivially bounded “for” loops).

The synchronous designed application can be considered as real-time if, for a particular execution architecture, the WCET of the step procedure is actually less than some deadline defined in the specification. Synchronous Programming and WCET estimation can then be viewed as two complementary and orthogonal methods for the design of safe real-time application:

– Synchronous Programming focuses in functionality (and thus determinism) by abstracting away “timing duration details”, while guaranteeing by construction that a bound will necessarily exist.
– WCET analysis can be performed afterwards to find an actual bound and checks that the implementation is real-time.

The goal of this work is to enhance the WCET estimation by pruning execution paths that are infeasible due the semantics of the high-level design.

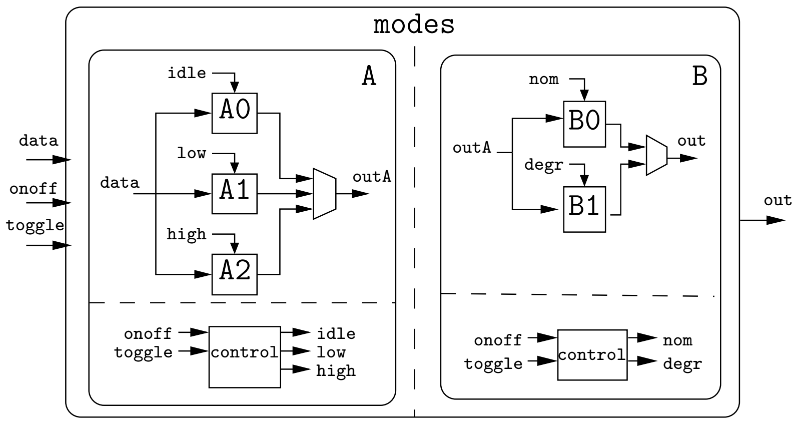

Fig. 1. Typical control system design.

2 Our approach on a realistic example

In this section, we illustrate our method, step-by-step on the typical synchronous design shown in Fig. 1. This application is made of two concurrent modules A and B. Each module is organized according to the *operating mode* paradigm: depending on the logical variables computed by the `control` part, the module computes exactly one particular mode (e.g. $A0$, or $A1$ or $A2$). These *intra-exclusion* properties, which can be formally proved on the high-level design, are not necessarily obvious at the binary level: it strongly depends on the programming language and its compiler. Moreover, for this particular design, we also know that there is a relation between the modes of B and the modes of A: whenever A is not idle (modes $A1$ or $A2$), B is necessarily in degraded mode ($B0$). This kind of *inter-exclusion* property has no chance to be obvious (i.e. structural) at the binary code, whatever be the language and the compiler.

The goal is to use this kind of properties to enhance the WCET. It requires to solve several problems:

– How to trace information between the high-level design and the binary code? More precisely, how to relate choices in the binary code to the high-level variables?
– How to find and prove the “right” properties, that is, the ones that are likely to prune infeasible paths?
– How to translate high-level properties into a suitable form for the Implicit Path Enumeration Technique? That is, in our case, how to express them as Integer Linear constraints.
– Finally, how to define a complete pruning algorithm with a good trade-off between the computation cost and the precision of the results? As a mater of fact, the techniques involved in this approach are well known to have a combinational cost: formal proof at the high level, ILP resolution...

2.1 Traceability

The compilation scheme of synchronous programs has two stages: the high-level design is compiled into an "agnostic" sequential language (C, most of the time), and then compiled into binary code.

Traceability between high-level and C code raises no problem: the code generator can be easily patched to associate to the C "if-then-else" statements the corresponding high-level variable. Traceability between C and binary raises a technical problem: patching an existing C compiler (gcc for instance) requires a lot of work, that is likely to be lost as soon as a newer version will be released. From a pragmatic point of view, we found more reasonable to rely on the "standard" debugging information provided by the compiler. This solution may lose information, especially when intrusive code optimization are used. However, we can expect it to be safe: either the debugging information clearly associate a binary branch to some C statement, or we simply ignore it.

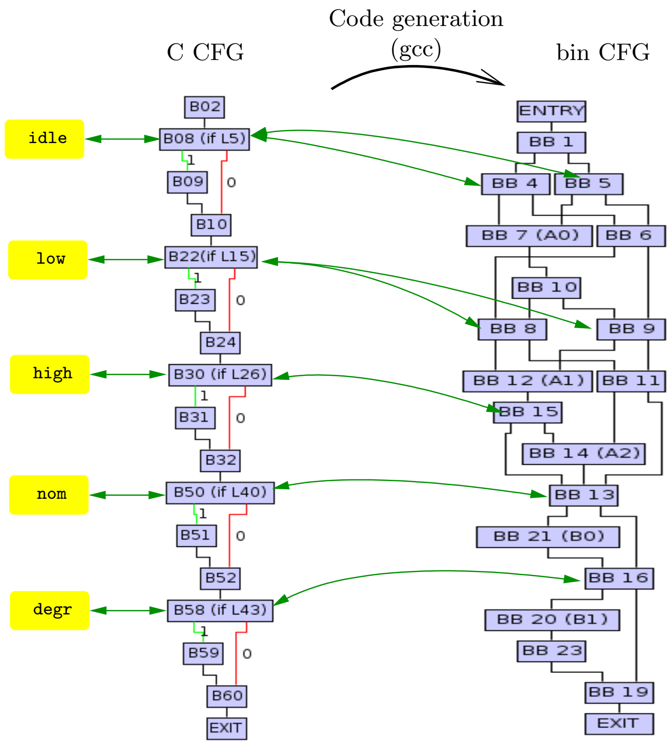

Fig. 2. CFG traceability with -O2

Fig. 2 shows the result of the traceability analysis for our example. Here, the code is obtained with the `-O2` optimization level of gcc, which is relatively intrusive and strongly modifies the control structure. Compared with the C structure, some test has been introduced, that are not clearly related to a C statement (e.g. block BB1); furthermore, some C statements have been "duplicated" (e.g. BB4 and BB5). Finally, the heuristic, based on the debugging information was able to relate 7 choices in the binary code to some binary choice in the C program, and then back to 5 high-level variables.

2.2 Checking infeasibility with a Model-Checker

Thanks to the traceability information, we can relate a conjunction of binary branches to a conjunction of high-level conditions. For instance, taking both branches BB4 to BB6 and BB13 to BB21 during an execution, requires that the high-level condition `''not idle and nom''` is satisfied. We can use a high-level verification tool to get information about this condition. We actually used Lesar, a model-checker dedicated to the Lustre programs. For instance, we can ask Lesar to prove that the Boolean expression `''not(not idle and nom)''` is an invariant of the program, that is, infinitely true along any execution of the program. In this case, the proof succeeds, meaning that the condition `''not idle and nom''` is actually infeasible, and thus, that all the corresponding execution paths can be removed for the search of the WCET.

2.3 Translating infeasibility into ILP

Taking infeasibility into account can be interesting for the micro-architecture analysis: it may help to enhance the analysis of features like instruction pipeline or memory cache hit and miss.

However, for this experiment, we are using this information only at the last stage of the WCET analysis: Implicit Path Enumeration. This supposes to translate the information (exclusivity of branches) into some Integer Linear constraints. This is particularly simple with the kind of code we are considering here: each branch is taken at most once, and thus, two branches are exclusive is their sum is strictly less than 2. For instance, the property `not(not idle and nom)` is translated into $\#BB6 + \#BB21 \leq 1$, where #BBi is the numerical variable encoding the number of time BBi is executed.

2.4 Putting all together: a proof of concept prototype

Several solutions have been explored to define a fully automatic enhancement algorithm. The first one is rather intuitive and behaves as a try and refute iterative algorithm:

– a first WCET path candidate is found using the standard ILP solving method,
– from this path candidates, according to the traceability information, we build a high-level infeasibility property, and ask the model-checker to prove that it is an invariant:

- if the proof succeed, the worst case candidate is proven infeasible, we translate this information into an additional Linear constraint, solve the system again to find another WCET candidate and so on,
- if the proof fails, it means that the WCET candidate is feasible (modulo the decision power of our tool), and the iteration stops.

This iterative algorithm eventually converges to a WCET estimation which is "optimal" (modulo the model-checker). However, the global cost is likely to be intractable, because the number of necessary iterations grows in a combinatorial way.

An alternative heuristic is based on the empirical remark that "interesting" properties are often simple pairwise exclusions. The idea is then to:

– identify the set of interesting high-level control variables (i.e. the ones that are related to binary choices),
– check the validity of all the pairwise relations between this variables,
– translate back all the proven relations into ILP, and solve the system once.

This solution is not guaranteed to provide an optimal solution. However, its cost is likely to remain reasonable: only a polynomial number of relations has to be considered (quadratic); each relation is "small" (involving only 2 variables) and is also likely to be proven or refuted with a reasonable cost.

At last, it is possible to define an extension and/or a mix of the methods, for instance:

– complete the pairwise method with some iterative steps to guaranty optimality,
– consider relations involving more than 2 variables (in which case they should be selected carefully in order to limit the number of combinations).

2.5 Some results and conclusion

We have developed a fully automatic tool, that orchestrate several external tools for the search for an enhanced WCET estimation. It uses:

– The language Lustre and its compiler (`lus2c`) for designing and compiling the high-level application [5].
– The compiler `arm-elf-gcc` to produce binary code (for the ARM architecture).
– The tool OTAWA for the micro-architecture analysis and to build a first ILP system [6].
– The tool LPSolve to solve ILP systems.
– The model-checker Lesar to prove invariants of the high-level (Lustre) code [7].

Figure 3 shows the general organization of the tool and its relations with external tools.

Table 1 gives some interesting quantitative results for the example of Fig 1, compiled without optimization. The interesting facts are the following:

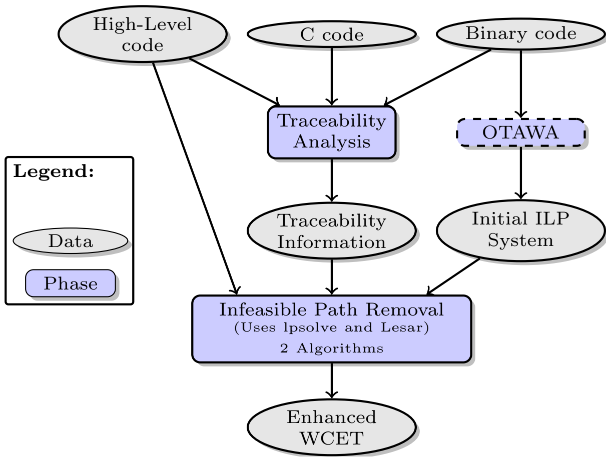

Fig. 3. Proof-of-concept Workflow

Wcet estimation		# iter.	ILP constraints	cost (cpu seconds)		
initial	final		(# / arity)	LPSolve	Lesar	Over. (Total)
4718	2371	298	298 / 30 var.	90s	5.2s	99s (163s)

Wcet estimation		Lesar (# pairs)		ILP constr.	cost (cpu seconds)		
initial	final	checked	valid	(# / arity)	LPSolve	Lesar	Over. (Total)
4718	2372	144	21	396 / 2	0.1s	2.2s	3s (67s)

Table 1. Some results with the 2 algorithms: iterative (top) and pairwise (bottom)

– As expected, the enhancement is important (about 50%).
– The (unavoidable) cost of OTAWA is 60 cpu seconds (not represented, we give only the overhead due to our method). This cost can be explained by the relatively important code size (83 basic blocks and 119 edges).
– The iterative method requires a huge number of steps (298), and as many calls to both LPSolve and Lesar. The cost of Lesar remains reasonable, while the cost of LPSolve "explodes": this is due to the number and the size of the extra linear constraints –in the last iteration, LPSolve handles 298 extra constraints with up to 30 variables in each.
– The global cost of pairwise method is much more reasonable. Lesar is called to check 144 pairwise relations, and 21 are proven invariant and translated into simple ILP constraints. LPSolve is called once for a neglectable cost. Note that the result misses the optimal one for one cycle (2372 vs 2371): the obtained path is actually not feasible, but for a "reason" that involves more than 2 conditions, and thus, that cannot be handled by the method.

As a conclusion, this work proposes a method to improve timing analysis of programs generated from high-level synchronous designs. 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 (high-level to C, C to binary). For that purpose, we use an existing model-checker for verifying the feasibility of execution paths, according to high-level design semantics Furthermore, our approach may work on optimized 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 timing analysis of the binary code. For a most detailed presentation, the reader may see the long version [8].

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. (TECS) **7**(3) (2008)
2. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: RTSS. (2006)
3. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **16**(12) (1997)
4. Halbwachs, N.: Synchronous programming of reactive systems. Kluwer Academic Pub. (1993)
5. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proceedings of the IEEE. (1991) 1305–1320
6. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An open toolbox for adaptive WCET analysis. In: SEUS. (2010)
7. Raymond, P.: Synchronous program verification with lustre/lesar. In Mertz, S., Navet, N., eds.: Modeling and Verification of Real-Time Systems. ISTE/Wiley (2008)
8. Raymond, P., Maiza, C., Parent-Vigouroux, C., Carrier, F.: Timing analysis enhancement for synchronous program. In: RTNS. (2013) 141–150