Noname manuscript No.
(will be inserted by the editor)

Performance Debugging of Esterel Specifications

**Lei Ju · Bach Khoa Huynh ·
Abhik Roychoudhury ·
Samarjit Chakraborty**

Received: date / Accepted: date

Abstract Synchronous languages like Esterel have been widely adopted for designing reactive systems in safety-critical domains such as avionics. Specifications written in Esterel are based on the underlying "synchrony hypothesis", which needs to be validated when Esterel specifications get compiled to real implementations (such as C code). In this work, we present a model-driven and architecture-aware timing analysis framework for C code generated from Esterel and executed on general-purpose processors. By integrating model-level information into the traditional timing analysis, we can efficiently compute accurate time estimates via systematically eliminating a large number of infeasible paths in the generated code. Experimental results show that with our proposed intermediate representation level infeasible path analysis in the model compilation, we obtain up to 16.1% tighter WCET estimates compared to the traditional assembly code level infeasible path detection with substantially less analysis time. Furthermore, by maintaining the traceability links between Esterel specifications and the generated C code, we are able to map the time-critical computations at the C-level back to the Esterel-level.

Keywords Esterel · Synchronous programming · Performance Debugging · WCET analysis

L. Ju (🖂)
School of Computer Science and Technology, Shandong University
E-mail: julei@sdu.edu.cn

B. K. Huynh
Department of Computer Science, National University of Singapore
E-mail: huynhbac@comp.nus.edu.sg

A. Roychoudhury
Department of Computer Science, National University of Singapore
E-mail: abhik@comp.nus.edu.sg

S. Chakraborty
Institute for Real-Time Computer Systems, TU Munich
E-mail: samarjit@tum.de

1 Introduction

For safety-critical domains, synchronous programming [4,34] has always been considered a clean formalism for programming reactive systems, which exhibit a high degree of concurrency but call for deterministic and predictable execution. Languages based on this paradigm – such as Esterel [10], Lustre [20] and Signal [5] – assume that time is partitioned into discrete instants or clock ticks and the computation/communication for processing all events that occur within one clock tick happen instantaneously (i.e. in zero time). The resulting semantics — with concurrent threads running in lockstep — take care of all scheduling issues, thereby simplifying the task of programming and making such specifications amenable to formal verification/certification. Generating implementations directly from synchronous language specifications is widely practiced in safety-critical domains such as avionics where certification of the generated implementation is essential.

A real-life implementation generated from a high-level synchronous program can be said to follow the synchrony hypothesis if all events that are logically assumed to be processed instantaneously are processed before the next set of events arrive. Verifying the synchrony hypothesis when a synchronous language program is compiled into hardware is relatively straightforward. As a result, compiling synchronous language specifications directly into hardware is currently the most popular design flow ([7]).

On the other hand, when synchronous languages are compiled into software – e.g., into a sequential C code – validation of the synchrony hypothesis is more complicated and depends both on the generated code, as well as on the micro-architecture of the platform executing this code. For the synchrony hypothesis to hold, the estimated Worst-case Execution Time (WCET) associated with the processing of events should be less than the minimum separation time between the arrival of sets of events (that are assumed to be processed instantaneously). Currently, such a systematic design process is missing in the context of software implementation of synchronous languages when the target platform is a general-purpose processor. This has primarily been due to the lack of mature software timing analysis techniques for general-purpose processor architectures. However, recent advances in WCET analysis techniques and the availability of industry-strength tools ([17,44]) has renewed the interest in synchronous language-based design flows targeting general-purpose platforms. Existing WCET analysis methods and tools can be directly applied to the C/assembly code generated from a high-level model for WCET computation. However, they can be customized to capture the characteristics of the modeling language and model-to-C compilation technique, which results in more efficient and accurate WCET estimation.

Our Contributions: The ultimate goals of this work include

– enabling software execution of Esterel specification on general-purpose processors. We apply WCET analysis techniques to validate the "synchrony hypothesis" of the generated C program from an Esterel specification.

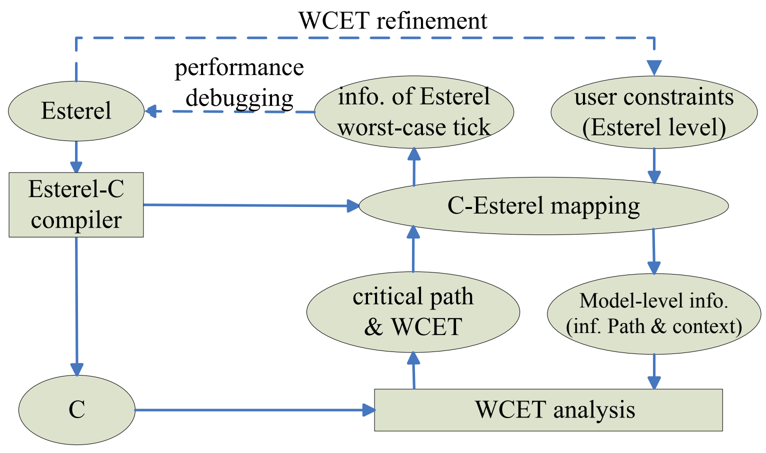


Fig. 1 Timing analysis framework for Esterel specifications.

– obtaining tight WCET estimates by exploiting model-level information in the code-level WCET analysis. Tighter WCET estimates clearly lead to more cost-effective and resource-saving designs, which is of considerable interest in the cost-sensitive application domains, such as automobiles.
– providing timing feedback to system designer for performance debugging and WCET refinement. By analyzing the results from WCET analysis, potential performance bottlenecks are identified and better resource dimensioning can be achieved.

In this paper, we propose comprehensive timing analysis techniques for general-purpose single-processor execution of C programs generated from Esterel specifications. Figure 1 presents an overview of our timing analysis framework. We build a bi-directional C-Esterel mapping by instrumenting the well-known Columbia Esterel Compiler (CEC) [13], which provides traceability between Esterel specifications and generated C programs. As a result, we can automatically feed model-level information to code-level WCET analysis for tighter WCET estimation. Furthermore, it allows us to highlight the Esterel-level critical path (statements executed in the single tick that result in WCET). Based on such timing feedbacks, system designer can either optimize the Esterel specification, modify the system configuration, or refine the WCET analysis by providing Esterel-level user constraints (e.g., environment constraints on input signal relationship) if the critical path is found to be an infeasible execution.

We extend existing WCET analysis techniques for general C programs to obtain WCET estimates on the C programs generated from high-level Esterel specification. The main novelty of our model-driven WCET analysis stems from two observations. First, the identification and removal of infeasible paths in the generated code can exploit the syntax and the semantics of the source Esterel specification. In particular, in our model-driven timing analysis framework, the infeasible path detection is performed during the Esterel-to-C compilation, instead of the assembly code level detection in traditional WCET analysis. This simplifies the problem to a large extent, especially when the automatically generated code usually contains huge number of infeasible paths compared to hand-written programs. Second, performing the analysis at a higher level allows us to easily identify infeasible paths resulting from the context-sensitive tick transition execution, which can be hardly done in the low-level analysis where the control flow information is encoded by data manipulation across function boundaries. Our experimental results show that up to 32% tighter WCET estimation can be achieved with our proposed infeasible path elimination, which improves the accuracy of the synchrony hypothesis validation, and provides system engineers with more flexibility in terms of design choices. Furthermore, comparing to traditional WCET analysis with assembly code level infeasible path detection, our proposed framework obtains up to 16.1% tighter WCET estimation with substantially less analysis time.

In this paper, we focus on timing analysis of Esterel specifications on *general-purpose* processor architectures. We rely on the traditional analysis techniques for WCET calculation, including program control flow analysis (i.e., the implicit path enumeration technique, or IPET [27]), and complex microarchitectural modeling (e.g., the cache persistence analysis [16], and out-of-order pipeline modeling [25]). On top of the traditional WCET analysis, our proposed timing analysis framework introduces additional model-level and model-to-C compilation information for a more efficient and effective WCET estimation. As a result, our proposed framework is not restricted to a particular micro-architecture. Furthermore, the infeasible path patterns and execution context information in the Esterel specification with CEC compilation can also be observed in many other concurrent finite state modeling languages and control flow graph-based model compilation (e.g., the MATLAB Stateflow model), which suggests a wider applicability of our proposed model-driven timing analysis framework.

2 Related Work

The synchrony hypothesis provides strong semantic soundness and ensures deterministic behavior. As a result, it is widely used in in practical embedded system design. Examples of programming models based on the synchrony hypothesis are Esterel [10], Lustre [20], Signal [5], and SyncCharts [2]. Recently, there are proposals for synchronous extensions of C language, including the Precision Timed C (PRET-C) [37,23] and SyncCharts in C (SC) [43]. In order to enable software execution of synchronous programs (or low-level executables compiled from synchronous programs), timing analysis must be performed to ensure the synchrony hypothesis holds in the real implementation. In this section, we briefly summarize the existing work on timing analysis of synchronous programs.

2.1 High-level WCET analysis

Architecture independent high-level timing analysis of Esterel (or Esterel-like) programs have been studied in [8,39,29]. For example, [29] employs model checking techniques to perform a high-level WCET analysis for the synchronous language Quartz, which is similar to Esterel. Given a synchronous language specification, the task of a high-level WCET analysis is to compute the worst case computation time for a particular input event (or all allowed inputs), in terms of the number of clock ticks required. This is usually done by translating the synchronous specification into a finite-state machine whose transitions correspond to clock ticks in the model. In other words, the *high-level* WCET analysis problem is concerned with the number of Esterel clock ticks, rather than the execution time of code within a clock tick.

The timing analysis problem where the states of the automata have been annotated with WCET estimates has been discussed in [31]. Again, the focus here was not to obtain tight WCET estimates, but to analyze high-level timing properties of an Esterel specification, assuming that platform-level WCET estimates are already available.

2.2 Code-level WCET analysis

Code-level WCET analysis for the synchronous models aims to find architecture dependent execution time for computation within a single clock tick in the generated low-level executable code (e.g., C or assembly). [30] performs the low level WCET analysis for the synchronous language Quartz by building a formal transition model on the statements in the generated executable code. Transitions are labeled with the physical execution time of corresponding statements, and symbolic model checking is applied to search the "longest" WCET path. However, the presented technique is only applicable to automata-based code generation, which does not scale well for large Esterel programs.

The problem addressed in [36] is the closest to what we study in this work. Here, the problem of infeasible paths in the generated code is mentioned and timing analysis of the whole Esterel program is studied. Though the work can also be used for estimating the maximum computation in a clock tick, the methodology is restricted, since it requires two separate codes to be generated from the synchronous program — one on which the WCET analysis is performed, and one which guides the analysis. The approach in [36] is only feasible for the generation of circuit code, which tends to be slow for large-scale application specifications. Furthermore, the problem of bidirectional traceability or performance debugging of Esterel specifications – even though mentioned – was not studied on non-trivial Esterel benchmarks by including traceability links in an Esterel compiler.

[40] reports practical experiment results on WCET analysis of avionics programs, including a program that is automatically generated from a synchronous language SCADE ([38]). A WCET analysis framework that integrates

the SCADE development environment from Esterel Technologies with the aiT WCET analyzer from AbsInt GmbH [1], targeting general-purpose processors, was presented in [21]. In [21], the WCET analysis is ignorant to the fact that the executable code is compiled from a high-level modeling language. On the other hand, our proposed model-driven timing analysis framework automatically utilizes model-level information in low-level WCET analysis, which leads to more efficient and tighter WCET estimation. Very recently, [42] proposes a technique to improve timing analysis for MATLAB Simulink/Stateflow model, by incorporating model-level flow information into WCET analysis. However, the model-level flow constraints are manually identified and translated into code-level flow constraints for WCET calculation.

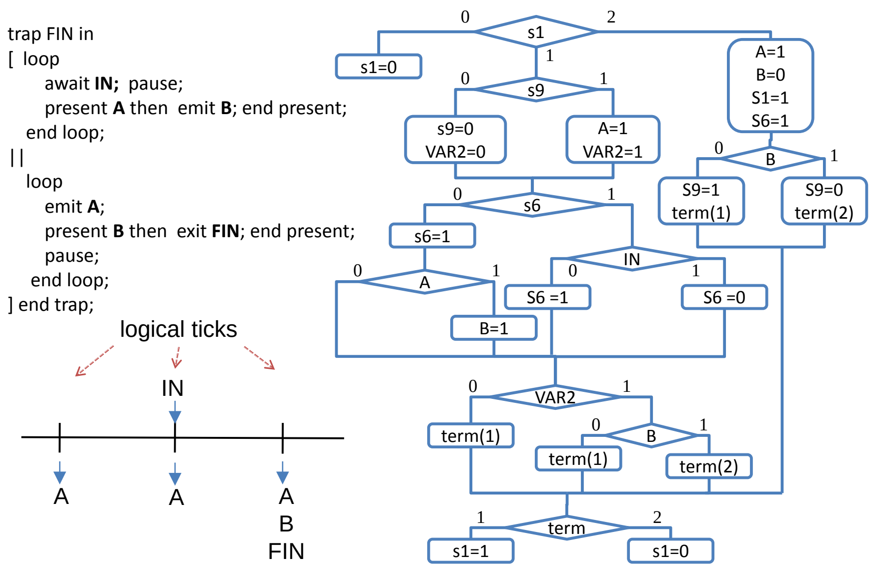
2.3 Timing analysis for special-purpose architectures

Special-purpose reactive processors have been developed to support concurrent execution of Esterel specification, where instead of compiling into C code, the Esterel specification is mapped to a concurrent reactive processing ISA. Example architectures include EMPEROR [45], STARPro [46] and Kiel Esterel processor (KEP) [26]. Timing analysis for execution of Esterel specifications on the special-purpose reactive processors have been studied in [24,45,9,32]. Recently, timing analysis techniques based on model checking and reachability analysis have been proposed for a synchronous version of C, called Precision Timed C (PRET-C), to be implemented on special precision timed or PRET architectures ([37,23]).

Compared to timing analysis of general-purpose processors, architectural modeling for such special-purpose processors is simplified to a large extent. For example, order-of-order execution and caches are usually not integrated in these special-purpose processors. Furthermore, special-purpose reactive processors implement hardware supports for handling tick-based execution, concurrency, thread scheduling, and other Esterel semantics (e.g, preemption and event broadcasting). As a result, timing analysis designed for these architectures can not be applied to the general-purpose processor setting.

3 Overview of Esterel

Synchronous languages like Esterel have been widely adopted for designing reactive systems in safety-critical domains such as avionics and automobiles. Specifications written in Esterel are based on the underlying "synchrony hypothesis", where all computation and communication, unless explicitly paused (using a `pause` statement), happen instantaneously. A run of a program typically consists of steps or *reactions* in response to *ticks* of a global clock. With each clock tick, a reaction computes the values of output *signals* and a new state from the input *signals* and the current state of the program. Such a reaction completes (in zero time) if it does not contain any `pause`, or else it delays the instructions following the `pause` until the next clock tick.



```
trap FIN in
[  loop
      await IN;  pause;
      present A then  emit B; end present;
   end loop;
||
   loop
      emit A;
      present B then  exit FIN; end present;
      pause;
   end loop;
] end trap;
```

Fig. 2 An example Esterel program and its SCFG

For example, the program "`emit A; emit B; pause; emit C; pause; emit D`" *emits* the signals `A` and `B` at the first tick, `C` at the second tick, and `D` at the third tick. If `p` and `q` are Esterel statements, then `p` $\|$ `q` is the parallel composition where `p` and `q` are executed concurrently with signals between `p` and `q` being transmitted instantaneously. We use a toy Esterel program with two concurrent threads shown in Figure 2 as the running example in this paper. The first thread blocks till the input signal IN appears, and passes the control to the "present A" statement in the next clock tick (after the "pause" clause). The second threads emits a signal A in each clock tick, and passes the control to the end of the "trap" block if signal B presents. A sample execution trace is shown in the Figure 2, where the input signal IN appears in the second tick, and the "trap" block terminates in the third tick. Further details of the syntax and semantics of Esterel may be found in [10] (or from the references in [4]).

Compiling Esterel. Esterel programs can be compiled into C programs to be simulated/executed on general processor architectures. In principle, the generated C code should preserve the semantics of original Esterel program by

– implementing a *tick function*, such that one complete execution of the function (between its entry and exit) represents Esterel computation and communication required to be instantaneously executed within one clock tick. The tick function is loop-free, since Esterel allows no loops within a clock tick.
– encoding the automata of tick transitions within the tick function, which preserves the context information of the clock tick, and determines the path to be executed in the tick function.

– sequentializing the concurrent execution within a tick, based on the control dependencies (e.g., clock tick boundary, preemption) and communication dependencies (between set and test of signals) defined in the Esterel program.

Various techniques exist for compiling Esterel into C programs [35]. Based on the intermediate representation used, they can be categorized into automata-based, netlist-based, and control flow graph (CFG) based approaches. An automata-based Esterel compiler usually generates a separate branch for each possible state (representing a clock tick) in the specification. The generated code is very fast to run, with very small overhead to determine the state to be executed. However, the size of generated code grows exponentially with the number of concurrent threads in the specification. Netlist-based approach translates each Esterel statement into a netlist of boolean logic gates. No statement duplication is required in the generated code, which leads to much more compact code. However, the main drawback is the significant increase in execution time. In this paper, we will focus our discussion on the control flow graph-based Esterel compilation, which normally produces fast and small C code. Instead of doing a comparison of various compilation techniques, in this paper we study how to compute a tight WCET estimation if the CFG-based compiler is adopted for a particular application. CFG-based compilation complicates the WCET estimation to a large extent, due to the complex control flow in the generated code. Our ultimate goal is to utilize the model-level and compilation information to guide the low-level analysis for an efficient and effective WCET estimation.

As mentioned before, we have integrated our work into the control flow graph-based code generation of the Columbia Esterel Compiler (CEC) [13]. CEC first parses an Esterel program to build an abstract syntax tree (AST), which is then used to generate a variant of the so-called Graph Code (GRC) [35] through a syntax directed translation. GRC represents a concurrent structure of the desired cycle function and uses a selection tree to encode the transition between cycles. It is an elegant way to represent the Esterel program, which allows optimizations to be performed prior to C code generation. The GRC is then transformed into a sequential control flow graph (SCFG), via a set of intermediate representations like the program dependence graph (PDG), and the concurrent control flow graph (CCFG). In CEC, these intermediate steps ensure that the concurrent control flow in GRC is sequentialized with the minimum number of context switches, while obeying the control/data dependencies in the original Esterel program. Finally, sequential C code can be directly generated from the SCFG.

The SCFG generated by CEC for our running example is also shown in Figure 2 (with some minor modification to make it compact). In the SCFG, multiple outgoing edges from a source node represent conditional branches, and the value for the branch to be taken is also labeled. It is common to have the following types of variables in the SCFG:

– signal variables, e.g., IN, A and B.

– variables to control the context switches between concurrent threads, e.g., $VAR2$.
– guard variables which handles the termination/preemption constructs in the Esterel, e.g., *term*.
– state variables which encodes the control flow of tick transitions, e.g., $s1$, $s6$, and $s9$.

The signal variables are usually of boolean type, representing the absence or presence of the signal in a particular tick. The last three types of variables are introduced by the CEC compiler to handle the control flow of the Esterel execution, which usually have finite integer values depending on the Esterel program structure.

The SCFG can be directly used to generate a C program with very similar control flow. One complete path in the SCFG and the corresponding C code (from the "test $s1$" to any of the sink node in our example) represents the execution of a possible Esterel tick. We will show that there can be a large number of infeasible execution path in the generated C program later in Section 5. By identifying the common infeasible path patterns, we are able to perform an efficient and effective SCFG-level infeasible path detection. Furthermore, as one can see that in the CFG-based Esterel compilation (similarly for CFG-based compilation of other finite state languages), it is not a trivial task for a programmer to map a C program path back to its corresponding Esterel tick execution. In Section 6, we will present a framework which automatically provides bi-directional traceability between the Esterel specification and the generated C program.

4 Overview of WCET Analysis

Static worst-case execution time (WCET) analysis computes the maximum execution time of a program on a micro-architecture for all possible inputs. We now give a brief overview of WCET analysis techniques for sequential programs. WCET analysis of a program involves finding the "longest" execution trace in the program's control flow graph (CFG). Recall that the nodes of a CFG are the basic blocks (maximal code fragments which are executed without control transfer), and the edges denote control transfer between basic blocks. Thus, a *path* in a control flow graph is simply a sequence of basic blocks, and an *execution trace* is a path executed for some program input. WCET analysis tries to find the maximum time the program takes to execute for any input. Figure 3(a) shows an example program and its control-flow graph.

Finding the weighted longest execution trace in a program can be done by running all possible inputs. However, this is not practical since (a) the number of inputs may be large, and (b) the program execution time for the same input may be different on different processors. WCET analysis methods typically solve this problem by developing a *static analysis* framework which takes as inputs (i) the program P being analyzed and (ii) a processor platform

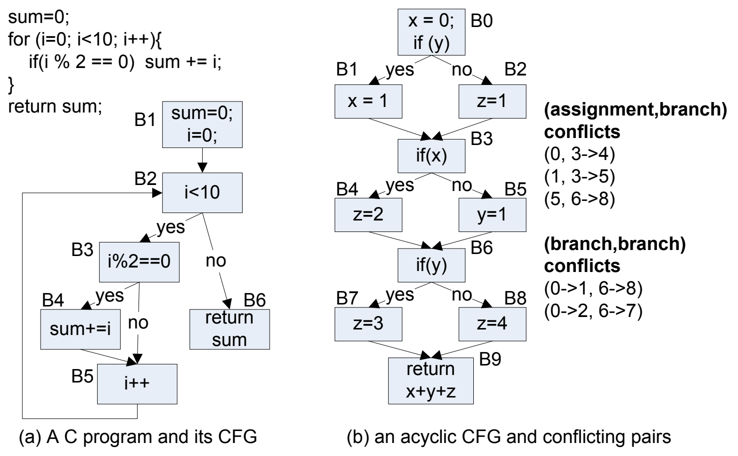


Fig. 3 Two example control flow graphs.

description *Proc*, and produces as output an *overestimate* of the WCET of program *P* on processor *Proc*.

Static analysis based WCET estimation proceeds by finding the longest path in the program's (assembly code level) control flow graph, satisfying certain loop bounds (*e.g.*, in the example of Figure 3(a) the loop bound for the only loop is 10). There many existing static WCET analysis methods and tools for general programs and processor architectures (refer to [44] for a survey). In particular, the Integer Linear Programming (ILP) formulation can be used in low-level (e.g., assembly code level) WCET analysis, so that no explicit path enumeration is required (which is not feasible for complex program structure). In this work, we adopt an ILP-based WCET analysis framework, and take advantage of its flexibility to extend the framework for our language-specific performance analysis.

In an ILP-based WCET analyzer, the execution time estimate of each basic block is found by micro-architectural modeling where we develop timing models of the processor micro-architecture (*e.g.*, pipeline, cache, branch prediction) to find the WCET of a sequence of instructions. Note that the WCET estimate of the instruction sequence corresponding to a basic block *B* is an upper bound on the execution time of *B* under all possible execution contexts.

With the knowledge of WCET of the basic blocks, finding the WCET of the whole program is reduced to an optimization problem. Here, we maximize the program execution time *without* enumerating the execution traces of the program. This is done by expressing linear constraints on the execution counts of any node/edge of the assembly code level control flow graph. We then maximize an objective function representing the program execution time subject to these linear constraints. Since the execution counts of control flow graph nodes/edges are integers, we can employ the ILP technology. Formally, let $\mathcal{B}$

be the set of basic blocks of a program. The program's WCET is given as:

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is an ILP variable denoting the execution count of a basic block B and c_B is a constant denoting the WCET estimate of the basic block B. The linear constraints on N_B are developed from the flow equations based on the control flow graph. Thus for basic block B,

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ ($E_{B \to B''}$) is an ILP variable denoting the number of times control flows through the control flow graph edge $B' \to B$ ($B \to B''$). Additional linear constraints capture the loop-bounds (*e.g.*, in Figure 3(a) we need to add the constraint $E_{5 \to 2} \leq 10$).

4.1 Infeasible Path Detection

The core WCET estimation method outlined in the preceding is neither accurate nor automated. The cause of imprecision comes from the fact that many paths in the control flow graph might be *infeasible*, that is not appearing in the execution trace for any input. For example in the acyclic CFG shown in Figure 3(b), the execution path ($B0 \to B2 \to B3 \to B4$) cannot be taken for any program input, due to conflict between the assignment $x = 0$ (in $B0$) and the conditional branch $B3 \to B4$ (which can be taken only if $x \neq 0$). It is clear that undue WCET overestimation is introduced if an infeasible path is considered to be the longest path in WCET analysis.

Many techniques have been proposed to detect and eliminate infeasible paths at source/assembly code level for WCET analysis (e.g., [41,19]). In this work, we adopt a light-weight infeasible path detection technique based on the notion of *conflicting pairs* [41] — pairs of (assignment, branch) or (branch, branch) statements which may not appear together in an execution trace. Simply put, an assignment a on a variable `x` conflicts with a branch edge e (a branch edge refers to a branch condition being evaluated to either true or false) testing the same variable `x` if and only if (i) the test on `x` in e never succeeds with the value assigned in a, and (ii) there exists at least one path in the control flow graph between a and e which does not modify variable `x`. Similarly, a branch edge $e1$ testing a variable `x` conflicts with another branch edge $e2$ testing the same variable `x` if and only if (i) the conditions on `x` in $e1$ and $e2$ can never succeed together, and (ii) there exists at least one path in the control flow graph between $e1$ and $e2$ which does not modify variable `x`. Note that infeasible paths spanning across loop iterations are not captured by the definition of conflicting pair. Thus, [41] considers the control flow graph (CFG) to be a directed acyclic graph (DAG), representing the body of a loop. However, as we have discussed in Section 3, code generated from

Esterel specification (the tick function) contains no loop within execution of a single clock tick. Thus, we do not detect infeasible paths spanning across loop iterations.

The notion of conflicting pair is extensively used in this paper. To help readers have a better understanding the concept, we borrow the following formal definition of conflicting pairs from [41].

Definition 1 (Effect constraint) The effect constraint of an assignment $var := expression$ is $var == expression$. The effect constraint of a branch-edge e in the CFG for a branch condition c is c ($\neg c$) if e denotes that the branch is taken (not taken).

Definition 2 (Conflicting pair) A branch-edge (or assignment) x has a (branch, branch) (or an (assignment, branch)) conflict with a subsequent[1] branch-edge e if and only if

- there exists at least one path from x to e in the CFG, and
- conjunction of the effect constraints of x and e is unsatisfiable,

In Figure 3(b), we list the (assignment, branch) and (branch, branch) conflicting pairs in the example acyclic CFG. A conflicting pair captures a pair of statements which cannot be executed together *provided the variable resulting in the conflict is not modified in between the execution of these two statements.* In other words, a conflicting pair gets "invalidated" (no longer conflicting) if the effect constraint is modified in between of the two statements. For example, to exclude all possible infeasible paths that execute basic block $B0$ and branch $B3 \rightarrow B4$ without modifying x's value in between (basic block $B1$) from being considered as the critical path, we can add the following ILP constraint in the ILP-based WCET formulation

$$N_0 + E_{3\rightarrow 4} - N_1 \leq 1$$

where N_0, N_1 and $E_{3\rightarrow 4}$ are the 0-1 execution counts of basic block $B0$, $B1$ and branch $B3 \rightarrow B4$, respectively.

In general, conflicting pairs capture only pairwise conflicts, which cannot detect (and exploit) arbitrary infeasible path information. However, we will show that conflicting pair based infeasible path detection technique is efficient and effective for analyzing compiler generated code from high-level control-intensive models like Esterel.

5 WCET of a Single Tick

In this section, we propose a timing analysis technique for WCET estimation of a single Esterel tick. Such estimates can validate Esterel-level assumptions on the instantaneous processing of signals or events that occur together (Section 3).

[1] *Subsequent* in the sense of the topological order of the control flow DAG.

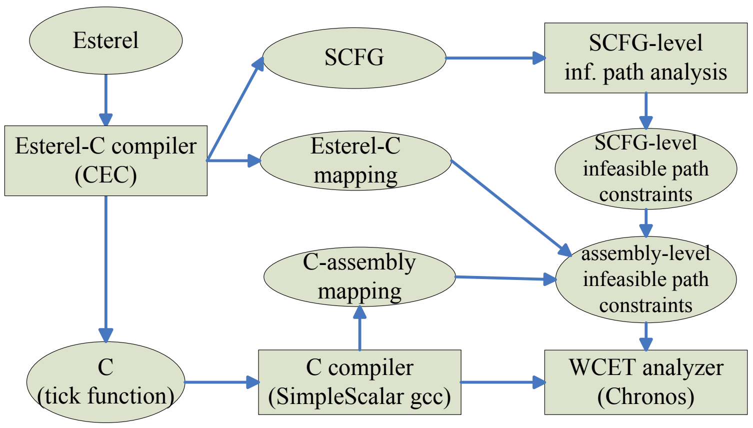

Esterel
Esterel-C compiler (CEC)
SCFG
Esterel-C mapping
SCFG-level inf. path analysis
SCFG-level infeasible path constraints
assembly-level infeasible path constraints
C-assembly mapping
C (tick function)
C compiler (SimpleScalar gcc)
WCET analyzer (Chronos)

Fig. 4 WCET analysis of a single Esterel tick.

Definition 3 **[Single tick WCET:]** For an Esterel specification to be compiled into a C program (the tick function) and executed on a given general purpose single-processor architecture, the maximum amount of computation and communication of the Esterel specification in *any* logical clock tick is bounded by the WCET of one single complete execution of the tick function (between its entry and exit).

Please note that in this paper, we focus on the CFG-based Esterel compilation ([13]). On the other hand, as described in Section 3, other compilation techniques generate code with much simpler control flow structure (at the cost of larger code size or slower execution), where the control flow analysis in WCET is straight forward. Furthermore, our motivation is to utilize model-level information for fast and accurate WCET analysis. Hence, we consider only the Esterel statements, and their corresponding C code. WCET estimation of the host functions written in general C programs can be done via the traditional analysis. For C code generated from Esterel specifications, the user can largely avoid the problems related to automation of the WCET analysis (refer to Section 4). In particular, since the tick function is loop-free (Esterel allows no loops within a clock tick), this leads to an acyclic control flow graph and hence there is no need to provide loop bounds to the WCET analyzer. Since each basic block is executed at most once in one execution of the tick function, an ILP-based WCET analysis produces a 0-1 assignment for the execution count of each basic block. However, compiler generated programs, especially from high-level control-intensive specifications, usually contain a huge number of infeasible execution paths compared to hand-written code. As a result, WCET overestimation due to infeasible execution paths is largely amplified in timing analysis of compiler generated programs.

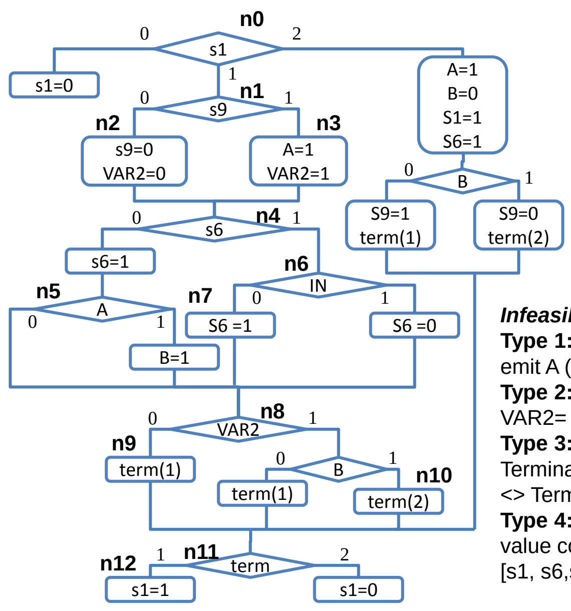
Figure 4 gives an overview of our WCET analysis framework. We use the Columbia Esterel Compiler [13]) to compile a given Esterel program into C, and calculate the WCET of the C code via an ILP-based platform-aware

WCET analyzer. We can validate that the synchrony hypothesis assumed at model level indeed holds in the real implementation, if the WCET of the generated tick function is guaranteed to be less than the minimum separation time between the arrival of sets of input events. We propose a comprehensive and light-weight infeasible path detection and elimination technique for WCET estimation of programs generated from Esterel specification. Our proposed infeasible path detection is performed at sequential control-flow graph (SCFG) level, which is a standard intermediate representation used in control-flow graph-based Esterel compilation. The computed SCFG-level infeasible path constraints are translated into assembly-level infeasible path constraints, via our Esterel-C mapping (obtained by instrumenting the CEC compiler) and the C-assembly mapping (obtained by disassembling the compiled C code). Finally, we integrate the assembly-level infeasible path constraints (as addition constraints) into the ILP formulation generated by Chronos ([15]), an ILP-based WCET analyzer. The WCET value and corresponding critical path for a single tick execution of the Esterel specification on a specific platform can be obtained by solving the resulting ILP formulation. In summary, we use the Esterel-level information and pattern of the generated tick function to identify infeasible path patterns, which are then taken into account during the timing analysis. Again, any C host functions in the Esterel specification, as well as the micro-architectural modeling, are handled via the traditional WCET analysis methodologies.

5.1 Infeasible Path Patterns

We observe that the automatically generated C code (from Esterel) often contains certain infeasible path patterns which may be less frequent in hand-written C code. Thus, low-overhead automatic methods for detecting/exploiting infeasible path information can substantially reduce the WCET of such automatically generated C code. Based on our study of the C programs generated via CDF-based Esterel compilation, we find the following four common sources of infeasible paths. We adopt the notion of conflicting pairs which has been presented in Section 4.1 in our discussion of infeasible path patterns. In Figure 5, we show the SCFG and the infeasible paths of the example Esterel specification in Figure 2. The four infeasible path pattern categorizations are as follows.

1. Emit and test signals. For example, the conflicts due to emit (set) the signal A (in SCFG node $n3$) and test absence of A (branch $n5 \rightarrow n8$). Besides, in an Esterel clock tick, the same signal may be tested in different concurrent threads. As a result, in the generated C program, multiple identical tests on the same signal variable will result in paths with (branch, branch) conflicts.
2. Sequentialization of concurrency in a tick. To generate sequential C code from a concurrent Esterel program, communication dependencies (between emit and test of a signal) and context switches between concurrent threads



Infeasible paths
Type 1: Emit and Test signals
emit A (n3) <> not present A (n5 - >n8)
Type 2: Sequentialization of concurrency
VAR2= 1 (n3) <> VAR2 ==0 (n8 - >n9)
Type 3: Termination and preemption
Termination code = 2 (n10)
<> Termination code == 1 (n11 - >n12)
Type 4: Encoding tick transition
value combination of control state variables
[s1, s6,s9] can *not* be [1, 1, 0]

Fig. 5 Example infeasible path patterns in generated C code.

must be captured. In CEC, this is handled by inserting new control variables and corresponding test nodes in the generated C code, when the concurrent control flow graph CCFG is translated into sequential control flow graph SCFG. In our example shown in Figure 5, the variable $VAR2$ captures the *state* of a thread before a context switch (by setting its value to 1 in node $n3$), and serves as a conditional guard and gets tested when the thread resumes execution at $n8$. Such assignments and tests (may be at multiple places in the same clock tick) on the guard will introduce possible infeasible paths.

3. Termination and preemption. The multi-threaded Esterel program follows the "wait for all threads to terminate" and "winner takes all" behaviors for thread completion and thrown exceptions ([13]). In the C code generated from CEC, this is handled by setting and testing the values of newly introduced guard variables (e.g. variable *term* as in the third example in Figure 5). These guard variables are assigned to non-negative integer values during the execution of each thread (0 for thread terminating, 1 for pausing, 2 and higher for throwing an exception). Such pairs of assignments and tests on these variables (e.g., $(n10, n11 \rightarrow n12)$ introduces possible infeasible paths.
4. Encoding tick transitions. In Esterel, a global automaton is defined on the sequence of ticks to be executed in each thread, via the use of "pause" and "await" statements. In the generated C code, this automaton is encoded through a set of state variables. Setting and testing these state variables introduce infeasible paths since certain combinations of states are not allowed

in the automaton. In our example shown in Figure 5, given the initial value of $s1$ to be 2, it is possible to have the value combinations of $[s1, s6, s9]$ to be $[2, \perp, \perp]$ ($\perp$ represents a undefined value), $[1, 1, 1]$, or $[1, 0, 1]$ before a feasible tick execution. On the other hand, $[1, 1, 0]$ is not a feasible state before any tick execution. As a result, corresponding paths are considered to be infeasible paths.

5.2 SCFG-level Infeasible Path Detection

In traditional WCET analysis, infeasible path detection is usually done via flow analysis at assembly code level [41,18]. In our previous work [22], we perform infeasible path elimination on C code generated from Esterel by capturing conflicting pairs at assembly code level. However, our previous experimental results in [22] show that due to large number of infeasible paths in the generated C code (several thousands of detected conflicting pairs), the analysis is complex and takes up to 15 minutes for the *mca200* benchmark from Estbench Esterel Benchmark Suite ([12]).

In this section, we propose a light-weight infeasible path detection at higher level during Esterel compilation. By maintaining a traceability link between model-level statements and compiled executables (the details will be discussed in Section 6), we automatically translate the high-level infeasible path information captured via conflicting pairs into assembly code level ILP constraints, which can be used in code-level WCET analysis to obtain tighter estimation results.

There are many levels of intermediate representations (IRs) while compiling an Esterel specification into assembly code for WCET estimate. For example, the control-flow graph based CEC compilation produces IRs including AST, GRC, PDG, CCFG and SCFG. In our work, we perform our infeasible path detection at SCFG level because of the following reasons.

- Any intermediate representations at higher level than SCFG does not contain all the infeasible path patterns. For example, the second type of infeasible path pattern due to sequentialization is only introduced when CCFG is translated into SCFG.
- C (assembly) code level analysis is incomplete without additional instrumented code. For example, the third type of infeasible path patterns will often produce many *switch-cases* constructs in the generated C code. The *switch* is translated into a *register indirect jump (jr)* in some ISAs [11], i.e. the branch target can not be determined statically.
- It improves the efficiency of conflicting pair detection. There are substantially fewer nodes in SCFG comparing to the number of assembly instructions. Moreover, no register/memory tracing or pointer analysis is required. Thus, the infeasible path analysis takes much less time at SCFG-level.
- As we will discuss in detail later, the state automaton construction for detecting the fourth type of infeasible path is obviously easier to perform at SCFG level than at any other lower levels (C or assembly).

Detection of infeasible paths type 1-3. The first three categorizations of infeasible paths listed in Section 5.1 are between pair-wise assignments and branches. To capture them in the SCFG, we adopt the notion of *conflicting pairs* [41] — pairs of (assignment, branch) or (branch, branch) statements which may not appear together in an execution trace (refer to Section 4.1). For example in Figure 5, node $n3$ (which assigns $A = 1$) and branch $n5 \rightarrow n8$ (to be taken when $A == 0$) is a (assignment, branch) conflict.

Conflicting pairs are easy-to-compute at SCFG-level. In Esterel semantics, a signal is either emitted or absent throughout any tick execution, corresponding to value 1 or 0 in the generated C code. Furthermore, domains of all compiler-introduced variables are statically known. We use the following notations in our discussion:

– DM_x is the domain of all possible values a variable x can be assigned to.
– $AS_{x,v}$ is the set of all nodes n in SCFG that contain assignment of x to value v.
– $BR_{x,v}$ be the set of all branch edges $e : n_i \rightarrow n_j$ in SCFG that are taken when $x == v$.

The (assignment, branch) and (branch, branch) conflicts on value v of variable x are represented as 4-tuple (n, e, x, v) and $(e1, e2, x, v)$. We compute the set of all conflicting pairs as follows.

$$\begin{aligned} AB = \cup_{v,v' \in DM_x} \{(n, n1 \rightarrow n2, x, v) | n \in AS_{x,v} \wedge (n1 \rightarrow n2) \in BR_{x,v'} \\ \wedge\, reach(n, n1) \wedge v \neq v'\} \\ BB = \cup_{v,v' \in DM_x} \{(n1 \rightarrow n2, n3 \rightarrow n4, x, v) | (n1 \rightarrow n2) \in BR_{x,v} \\ \wedge\, n3 \rightarrow n4 \in BR_{x,v'} \wedge reach(n2, n3) \wedge v \neq v'\} \end{aligned} \tag{1}$$

where $reach(n1, n2)$ is true if and only if there is a path from node $n1$ to $n2$ in the SCFG. For each (assignment, branch) conflict $(n, n1 \rightarrow n2, x, v)$ in AB and (branch, branch) conflict $(n1 \rightarrow n2, n3 \rightarrow n4, x, v)$ in BB, we also compute the set of nodes in SCFG that may "invalidate" the conflicting pair through assigning a different value to x. In other words, if x is assigned to a different value in between, the conflict between the assignment (test) on x in n $(n1 \rightarrow n2)$ and the test on x in $n1 \rightarrow n2$ $(n3 \rightarrow n4)$ is no longer valid. We compute the set of nodes that invalidate each conflicting pair as follows.

$$\begin{aligned} invalid(n, n1 \rightarrow n2, x, v) &= \{n' | reach(n, n') \wedge reach(n', n1) \\ &\qquad \wedge\, n' \in AS_{x,v'} \wedge v \neq v'\} \\ invalid(n1 \rightarrow n2, n3 \rightarrow n4, x, v) &= \{n' | reach(n2, n') \wedge reach(n', n3) \\ &\qquad \wedge\, n' \in AS_{x,v'} \wedge v \neq v'\} \end{aligned} \tag{2}$$

Computation of the above-mentioned sets AB and BB can be done in a single DFS traversal of the acyclic SCFG in $O(|N|+|E|)$ time. For each assignment of variable x visited, it will be pushed onto a stack SA_x. For each branch on x visited, it will be pushed onto another stack SB_x, and used to form pairs of (assignment, branch) and (branch, branch) conflicts with existing elements in SA_x and SB_x. During the backtracking, corresponding assignments and

State	State variable values	Next State(s)
st_0	$[s1 == 2, s6 == \perp, s9 == \perp]$	st_1
st_1	$[s1 == 1, s6 == 1, s9 == 1]$	st_1, st_2, st_3, st_4
st_2	$[s1 == 1, s6 == 0, s9 == 1]$	st_1, st_4
st_3	$[s1 == 0, s6 == 0, s9 == 1]$	
st_4	$[s1 == 0, s6 == 1, s9 == 1]$	

Table 1 Feasible States of the example SCFG shown in Figure 5.

branches will be popped off from the stacks. For each conflicting pair in AB or BB on variable x, finding the set of nodes that may invalidate it requires time $O(|N| \times |E|)$ to perform a reachability test for each node that updates value of x (captured in set AS). The set of conflicting pairs and corresponding "invalidating" nodes will be used to generate ILP constraints for infeasible path elimination (refer to Section 5.3). Compared to traditional low-level (e.g., assembly-code level) infeasible path detection, the proposed SCFG-level analysis is light-weight and much more efficient, due to the simplicity of the high-level abstraction. Associated experimental results are presented in Table 3, Section 5.4.

Detection of infeasible paths type 4. When an Esterel specification is compiled into C code, a set of state variables are introduced to encode the program execution context. Let's define a global state as a value combination of all the state variables. The number of state variables introduced in a control-flow graph based Esterel compilation (such as in CEC) usually depends on the number of concurrent threads in the Esterel specification, where each state variable captures the current tick of its corresponding thread (by having different values for each tick). For example, three state variables $[s1, s6, s9]$ are used in the SCFG shown in Figure 5.

Given the initial state and allowed finite state transitions defined by Esterel, certain value combinations of the state variables are *not* reachable. Our type 4 infeasible path pattern due to such unreachable combinations cannot be simply captured by the above-mentioned conflicting pairs, since (i) an infeasible path of this kind consists of many branches on state variable tests; and (ii) this type of the infeasible path patterns is "context-sensitive", i.e., a state variable's value in current tick depends on execution in previous tick.

Given a SCFG $scfg$ and its initial state st_0, Algorithm 1 shows how to find an *overestimated* set of all feasible states FS (i.e., the set contains *all* feasible states with possibly some infeasible ones). For a feasible state st, we compute feasible states st' that execute in the next tick after st. We consider all feasible paths in the SCFG by excluding the infeasible paths captured by conflicting pairs identified as in Section 5.2 (line 4). For a feasible path p, we first test whether p corresponds to the execution of the current analyzing state st (line 6-15). If any branch on a state variable s_i that can be taken when $s_i == v$ for a different value v from the defined value of s_i in st (line 8), the path p will not be considered when searching st's next states (line 13 -14).

Algorithm 1 $computeFeasibleState(scfg, st_0)$ — Compute set of all feasible states FS, where st_0 is the known initial state for SCFG $scfg$.

```
1: $FS.add(st_0)$;  $Queue.insert(st_0)$;
2: while $!Queue.empty()$ do
3:    $st = Queue.remove()$;
4:    for each feasible path $p \in scfg$ do
5:       $st' = st$;

6:       $curFlag = true$;
7:       for each branch $e$ on $p$ do
8:          if $e \in BR_{s_i,v} \wedge st[i] \neq \perp \wedge st[i] \neq v$ then
9:             $curFlag = false$; /*path $p$ is not in current state $st$*/
10:            $break$;
11:         end if
12:      end for
13:      if $!curFlag$ then
14:         $break$; /*search next path*/
15:      end if

16:      for each assignment node $n$ on $p$ do
17:         if $n \in AS_{s_i,v} \wedge st[i] \neq v$ then
18:            $st'[i] = v$;
19:         end if
20:      end for
21:      if $st' \notin FS$ then
22:         $FS.add(st')$;
23:         $Queue.insert(st')$;
24:      end if
25:   end for
26: end while
```

Otherwise, for each assignment that updates state variable s_i to a new value v (in the set $AS_{s_i,v}$) on p, we set the value in st' correspondingly (line 16-20). Finally, if the newly computed feasible state st' has not been visited before, we add it into the set of feasible state FS, as well as the workspace *Queue* so that states reachable from it will be computed in the future (line 21-24). Table 1 shows the list of all feasible states that are reachable from the initial state $[s1 == 2, s6 == \perp, s9 == \perp]$ (where $s6$ and $s9$ are undefined) given the example SCFG in Figure 5.

Let S be the set of all possible value combinations on all the state variables $< s_1, \ldots, s_n >$,

$$S = \prod_{i \in \{1,\ldots,n\}} DM_{s_i}$$

The set of all infeasible state IS can be obtained by

$$IS = S - FS \qquad (3)$$

where FS contains the set of all value combinations for reachable states as calculated in Algorithm 1. Note that we perform a conservative static simulation for reachable state calculation, which reports a superset of the "real" reachable states. We utilize the computed reachable states to obtain a safe subset of unreachable state variables' value combinations, and generate ILP constraints to eliminate the corresponding infeasible paths (see Section 5.3). The conservative simulation ensures that the WCET analysis never prunes any feasible program path and under-estimates the program execution time.

5.3 Infeasible Path Elimination

We now discuss methods to eliminate infeasible paths given the conflicting pairs and feasible state variables' value combinations. We generate an ILP constraint for each conflicting pair and infeasible value combination detected at SCFG level. As shown in Figure 4, we maintain an Esterel-C mapping which provides traceability between corresponding Esterel statements, SCFG nodes, and generated C statements. Together with the C-assembly mapping between C statements and basic blocks/edges in the CFG of the generated C code, we can *automatically* translate the SCFG-level ILP constraints into assembly code level ILP constraints. These assembly code level ILP constraints are utilized in an ILP-based WCET analyzer to prevent infeasible paths from being considered as the WCET critical path (thereby leading to a tighter WCET estimate).

Let $(n_i, n_j \to n_k, x, v) \in AB$ (or $(n_i \to n_j, n_k \to n_l, x, v) \in BB$) on variable x and its value v be a conflicting pair (Equation 1), and $invalid(n_i, n_j \to n_k, x, v)$ (or $invalid(n_i \to n_j, n_k \to n_l, x, v)$) be the set of nodes that invalidate it (Equation 2). Formally, we can encode this conflicting pair as a linear constraint

$$\begin{array}{ll} N_i + E_{j\to k} - \sum_{n_p \in invalid(n_i, n_j \to n_k, x, v)} N_p & \leq 1 \\ E_{i\to j} + E_{k\to l} - \sum_{n_p \in invalid(n_i \to n_j, n_k \to n_l, x, v)} N_p & \leq 1 \end{array} \quad (4)$$

where N_i ($E_{j\to k}$) is the 0-1 execution count of SCFG node n_i (edge $n_j \to n_k$).

For the fourth type of infeasible path pattern, we generate the following ILP constraints for each infeasible state $st : [s_1 == v_1, \ldots, s_n == v_n]$ in IS (refer to Equation 3).

$$E_1 + \ldots + E_n < n, \forall e_i \in BR_{s_i, v_i}$$

where E_i is the 0-1 execution count of edge e_i, for all $e_i \in BR_{s_i,v_i}$ that can be taken if state variable $s_i == v_i$ in the infeasible state st. In other words, we prevent the longest path to contain state variable evaluation corresponding to an infeasible state st. Hence, not all branches that can be taken in the infeasible state st are allowed to execute in a single tick.

Given the example in Figure 5, the ILP constraint

$$E_{0\to 1} + E_{4\to 6} + E_{1\to 2} < 3$$

will be generated to eliminate the infeasible path containing edges $n0 \to n1$, $n4 \to n6$, and $n1 \to n2$, which corresponds to an infeasible control state $[s1 == 1, s6 == 1, s9 == 0]$. Given the one-to-one mapping between nodes/edges in SCFG and basic blocks/edges in assembly code level CFG, the above constraints can be automatically translated into assembly code level ILP constraints for infeasible path elimination in WCET analysis.

benchmark	# of lines (Esterel/C)	conflicting pairs	WCET (cycles) w/o inf.	WCET (cycles) w/ inf.	WCET (cycles) reduction	sim. (cycles)	overest
runner	55/253	42	3905	3781	3.2%	3589	5.4%
reflex	96/378	105	5197	4971	4.4%	4649	6.9%
abcd	101/827	1796	10335	8463	18.1%	8099	4.5%
mejia	555/2598	5328	25983	23343	10.2%	18834	23.9%
tcint	687/3031	2848	13497	10949	18.9%	8869	23.5%
wristwatch	1088/1755	3307	40862	27773	32%	22300	24.5%
mca200	7269/10894	3402	129038	99396	23%	89541	11%

Table 2 WCET analysis results.

5.4 Experimental Results

We now present some implementation details and experimental results to evaluate our proposed WCET analysis for a single Esterel tick execution. We compiled Esterel programs into C using the default (control-flow graph based) code generation mechanism in the Columbia Esterel Compiler (CEC) [13]. We instrumented CEC so that during the compilation a C-Esterel mapping is created. We used Chronos [15], an ILP-based WCET analyzer, to calculate the WCET of the tick function in the generated C code. For the WCET analysis, the default architectural configuration of the tool was used, which assumes a direct mapped L1 instruction cache with 8-byte block size, dynamic 2-level branch predictor, 5-staged pipeline, and an instruction dispatch queue size of 4. We assume no data cache and the instruction cache miss penalty is 30 cycles.

We used benchmarks from Estbench Esterel Benchmark Suite [12], including a runner's behavioral description (*runner*), a simple combination lock (*abcd*), a shock absorber (*mca200*), and a *wristwatch* example.

Table 2 summarizes our WCET analysis results. For each program, we show the code size of the Esterel specification and the generated C program. The number of conflicting pairs automatically detected by our infeasible path detection algorithm are also listed. Comparing to normal hand-written programs, huge number of conflicts (infeasible paths) exist in automatically generated code. The analysis time spent on finding these conflicting pairs takes less than 1 second for each of the benchmarks. The calculated WCET values without and with the infeasible path detection for each benchmark are presented in column "w/o inf." and "w/ inf.". We can see 3.2% to 32% tighter WCET estimates can be obtained with our automatical infeasible path elimination technique. A tighter WCET value improves the accuracy of the synchrony hypothesis validation, and provides system engineers with more flexibility in term of design choices. Finally, we compare our WCET estimation (in "w/ inf") with the SimpleScalar ([3]) simulation results shown in column "sim." using the same architecture configuration. The potential WCET overestimation is presented in "overest". However, the simulation results are usually an under-estimation of the *real* WCET values. C programs generated from Esterel specifications are control-intensive programs that handle many concurrent input events in a single tick execution and have complex internal control states. Hence, worst-

Benchmark	asm-level analysis in Chronos V3		our SCFG-level analysis		
	WCET (cycle)	analysis time	WCET (cycle)	WCET reduction	analysis time
runner	3905	6.9s	3781	3.2%	0.008s
reflex	5197	11.8s	4971	4.4%	0.003s
abcd	8463	41.2s	8463	0%	0.02s
mejia	25238	2m15s	23343	7.5%	0.2s
tcint	13057	3m15s	10949	16.1%	0.5s
wristwatch	31278	2m34s	27773	11.2%	0.12s
mca200	113519	15m18s	99396	12.4%	0.82s

Table 3 Comparison with assembly code level infeasible path detection.

case inputs and program control flow contexts are difficult to identify in the simulation. As a result, the presented ratio only serves as an upper bound of the overestimation between our estimated WCET and the real WCET.

Assembly-level infeasible path detection. We have also compared our SCFG-level infeasible path detection with the built-in assembly code level infeasible detection in Chronos V3 (which is also based on the notion of conflicting pairs). The WCET results and infeasible path analysis time are shown under "SCFG-level" and "asm-level" in Table 3, respectively. As one can see, our proposed SCFG-level analysis outperforms the assembly code level infeasible path detection in terms of both accuracy and analysis time.

Compiler optimization. The above-mentioned experiments are based on WCET analysis of non-optimized assembly code (i.e., with *-O0* option in `gcc` compilation). The control-flow structure of a non-optimized assembly code is very close to that of SCFG in Esterel compilation. On the other hand, modern C compilers support optimizations to increase the performance of compiled code. For example, the `gcc` *-O3* option performs optimization techniques including dead code elimination, code-block reordering, function inlining, and register renaming. However, such optimizations may potentially increase the binary code size.

If the C code is compiled with optimizations, there may be large differences in the control-flow structures between the optimized assembly code and SCFG. As a result, (i) infeasible paths identified at SCFG level may not exist in the optimized assembly code, and (ii) the optimization may introduce additional infeasible paths which are not presented at SCFG level. However, any infeasible path detection must be sound, but may be incomplete. In other words, by simply discarding the SCFG-level infeasible path constraints that cannot be matched in assembly code level, we are still able to obtain a safe (but probably less accurate) WCET estimation.

Table 4 shows our experimental results with C compiler optimization. We show the number of infeasible path constraints obtained at SCFG level. We discard SCFG level constraints that cannot be matched at assembly code level (due to missing statements and/or branches). The number of "survived" constraints are shown in Table 4 for both *-O0* and *-O3* optimizations. In particular, the "asm(*-O0*)" column shows the number of SCFG-level infeasible

Benchmark	# of infeasible path constraints			*-O3* WCET (cycle)		
	SCFG	asm(*-O0*)	asm(*-O3*)	w/o inf.	w/ inf.	sim
runner	12	12	7	2681	2557	2304
reflex	41	28	13	4329	4171	3054
abcd	437	384	114	7605	6189	5395
wristwatch	2831	2278	512	28476	21925	13631

Table 4 WCET results with C compiler optimization.

path constraints that are successfully translated into assembly code level constraints when no optimization is used in C compilation. These constraints are used to produce the WCET results with SCFG level infeasible path constraints as shown in Table 2 and Table 3. On the other hand, the "asm(*-O3*)" column shows the number of SCFG-level constraints that can be translated into assembly code level constraints when *-O3* is used for gcc compilation. Finally, in column "*-O3* WCET (cycle)", we present the WCET estimations for *-O3* optimized assembly code, without and with the SCFG level infeasible path constraints, as well as the simulated WCET for each benchmark.

Compiler optimization can also be presented during Esterel-to-C compilation. However, since our infeasible path detection is performed at the SCFG level (usually the bottom-level representation where C code can be directly printed), model compiler optimization will not affect our proposed analysis.

6 Performance Debugging and WCET Refinement

In this section, we discuss how to achieve the bi-directional traceability and use it for performance debugging and WCET refinement of Esterel specifications, as shown in Figure 1. If the WCET estimate produced for the C-level tick function is greater than a pre-defined clock tick length, we have a violation of the synchrony hypothesis. It is then useful to show the programmer the Esterel statements executed corresponding to the WCET estimate. To provide such backwards traceability, a C-Esterel mapping is built during code compilation. This mapping is used to generate the Esterel-level critical path (statements executed when the WCET is realized) from the C-level critical path produced by the WCET analyzer. By visualizing these Esterel statements, the programmer can perform optimization/modification of the Esterel specification.

6.1 Building the Traceability

Assembly to C mapping. State-of-the-art WCET analysis tools typically perform the analysis on assembly code (which obtained by disassembling the program binary) rather than source code. This is to take into account the effect of compiler optimizations for accurate timing estimation. For an ILP-based WCET analyzer, the WCET estimate is given via basic block counts, where each basic block is a sequence of assembly instructions. Our first step towards maintaining backwards traceability is to provide a mapping from assembly to C code. This can be easily achieved by disassembling the C object file using the *objdump* command, which produces the link between assembly instructions and the corresponding C code.

C to Esterel mapping. To enable a mapping from the C-level WCET path back to the Esterel level, we maintain traceability links while compiling Esterel to C. In order to impose minimum overheads on the Esterel to C compilation, we only need to maintain C to Esterel mapping for a subset of Esterel statements. We only trace the Esterel statements that are eventually translated into C statements (such as data and signal processing, conditional statement, preemption statements, etc.) and affect the execution time of the generated C program. For Esterel statements that only affect the control flow of the C code and produce no explicit execution costs, we do not need to monitor them during the compilation process. In particular, we classify the Esterel statements into following four categories.

– Data and signal processing statements (e.g. expressions, assign, procedural call, emit). These statements need to be traced, because they are directly translated into C statements, and will explicitly affect the execution time of both the Esterel and C programs.
– Conditional and preemption statements (e.g. if-then-else, present-then-else, abort-when, trap-exit). We trace the predicate signal/expressions for these statements, which are translated into conditional tests in the C code. This is to reflect the time taken to evaluate and test these predicates. Furthermore, tracing these predicates helps to automatically generate constraints in WCET analysis, based on programmer's annotation given at Esterel level. We will discuss the details in Section 6.2.
– Other control flow statements (e.g. ‖, ;, loop, pause). These statements are translated into control flow in the generated C program. There are no explicit C statements which correspond to them. As a result, we do not trace these statements.
– Variable/signal declaration statements (e.g. signal, var, input, output). These statements are not traced since they are compiled into variable declarations in the C program.

When an Esterel program is compiled to C, it is first translated to an intermediate representation (IR), *e.g.*, the abstract syntax tree (AST). During the AST construction, we maintain a mapping from Esterel line numbers to the IR node ids. Subsequently the AST is transformed into a sequential control flow graph (SCFG) which sequentializes Esterel's concurrency. However, the computation/predicate nodes of the AST that we trace are retained in the SCFG. Hence, we can map the AST nodes to SCFG nodes. A mapping between IR node ids and C line number is created when SCFG is translated into C program. As shown in Figure 6, by combing all above-mentioned mappings, we can construct a mapping between the assembly code and Esterel specification.

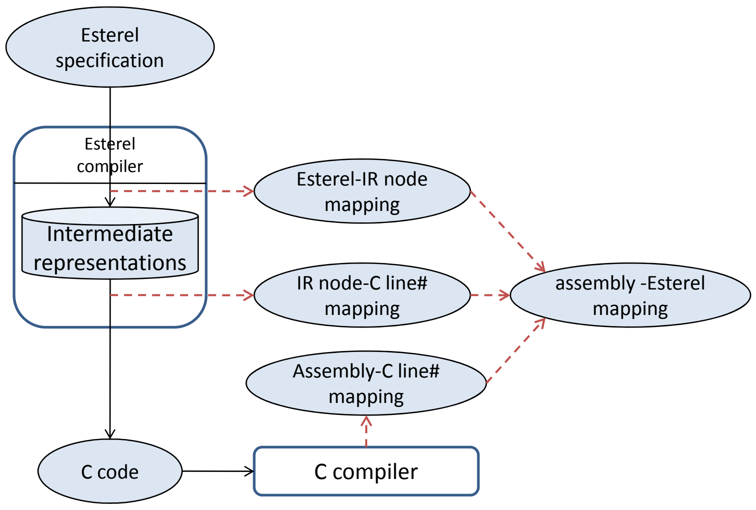


Fig. 6 Construction of the assembly-Esterel mapping in Figure 1.

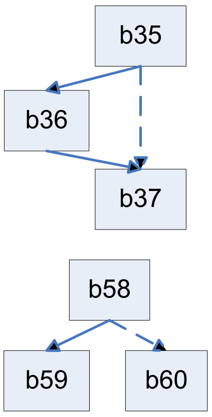
Mapping back the longest path. Recall that the ILP-based WCET analysis (as discussed in the preceding section) only reports the WCET estimate; it does not produce the corresponding longest path (also called the *critical path*). However, the control flow graph of the Esterel tick function is a directed acyclic graph or DAG and each basic block is executed at most once. C statements executed in the critical path of the tick function can be reconstructed easily from the 0-1 assignments of the basic block counts via the assembly to C mapping (*any C statement appearing in a basic block with execution count 1 must lie on the critical path*). Finally, via our C to Esterel mapping, the Esterel statements corresponding to the WCET can be obtained. However, since infeasible path detection methods are incomplete, the reported critical path may, in principle, still be an infeasible path. Hence, we allow the programmer to provide infeasible path annotations at the Esterel level. These are automatically translated into ILP constraints on the execution counts of the C program's basic blocks via our traceability links between Esterel, C and the assembly code.

What kind of infeasible path annotations can be provided at the Esterel level? Esterel allows the programmer to explicitly define # (exclusion) and => (implication) relations on signals. These are constraints on the environment of the Esterel specification (*e.g.*, signals x and y never happen in the same tick) which are automatically translated to ILP constraints for tighter WCET analysis. We have also extended the #, => relations to Esterel statements and predicates. In particular, we have defined two relational operators, ## (*conflict*) and <=> (*coexist*), between Esterel statements/predicates (represented using their line numbers) that we trace when building the C-Esterel mapping. These annotations can be automatically translated into ILP constraints as follows. A *conflict* annotation $A\#\#B$ is translated into the linear constraint $N_A + N_B \leq 1$ and a *coexist* annotation $A <=> B$ is translated into

```
1   module reflex_game
    …
7   relation ..., READY # STOP
    …
14  every COIN do
    …
22    [
23        copymodule AVERAGE
24            ||
          …
38            trap END_MEASURE in
39            [
40                every READY do
41                    emit RING_BELL
42                end
43                ||
        …
52      do
53          do
54              every MS do
55                  TIME:=TIME+1
56              end
57              upto STOP;
58              emit DISPLAY;
59              emit INC_AVE(TIME)
60          watching LIMIT_TIME MS
61          time out exit ERROR end;
62          emit GO_OFF;
63          exit END_MEASURE
64      ]  %trap END_MEASURE
            ...

87  module AVERAGE
    …
91    every immediate INC_AVE do
92      TOTAL := TOTAL + ?INC_AVE
93      NUM := NUM +1;
94      emit AVE_VALUE (TOTAL/NUM)
95    end
    …
```

Fig. 7 The reflex game Esterel specification and highlighted critical path.



```
b35:  if (READY) {
b36:    RING_BELL = 1;
b36:  }
b37:  …
…
b58:  if (STOP) {
b59:    DISPLAY = 1;
b59:    GO_OFF = 1;
b59:    _term &= -(1 << 2);
b59:    (INC_AVE_v = TIME),
            (INC_AVE = 1);
b59:    }
b60:    else {…
```

Fig. 8 C-level critical path (in solid lines) of the reflex game.

the linear constraint $N_A = N_B$, where $N_A(N_B)$ is the execution count of the basic block that contains $A(B)$ if $A(B)$ is a statement, or the execution count of the corresponding branch edge (evaluating to true) if $A(B)$ is a predicate.

6.2 Case Study in Performance Debugging

We illustrate our timing analysis framework using the well-known reflex game example [6]. A user can start a game session by inserting one COIN to a machine. To test his reflex time, once the user is ready (by pressing the *READY* button), he needs to press *STOP* as quickly as possible after the machine generates a *GO_ON* signal to turn on a light. This is repeated three times and finally the average reflex time will be calculated and displayed before game is over. Figure 7 shows an Esterel fragment of the game controller. The complete Esterel specification of the game can be found in [6] (game version 1).

We used CEC to compile the reflex game program. We instrumented CEC to produce a C-Esterel mapping as discussed earlier. Automated infeasible path detection and ILP-based WCET analysis were performed on the generated C code. Once the critical path was computed at the C level, we identified the critical path at the Esterel level via backwards traceability. Figure 8 shows a CFG fragment of the reflex game example, where assembly code level basic blocks in the critical path (blocks which have an execution count of 1) computed by our WCET analyzer are highlighted. Figure 8 also shows the corresponding C code fragment, through the assembly to C mapping. Finally, via the C to Esterel mapping, the corresponding Esterel statements executed in the worst case path are obtained. Thus, the C code fragment shown in Figure 8 corresponds to Esterel lines 40-42, 58-63 in Figure 7. The generated C tick function is acyclic, where signal emissions are translated to assignments of corresponding variables. Note that exception handling (e.g., trap END_MEASURE) are translated into the manipulation of compiler-introduced variables in CEC, which are not shown in the generated C code in Figure 8.

The entire Esterel level critical path is highlighted using shaded lines in Figure 7. It corresponds to the user pressing *READY* and *STOP* buttons simultaneously after the machine generates a *GO_ON* signal. In such a case, the machine rings a bell (*emit RING_BELL*) to indicate that the *READY* button is pressed wrongly (it should only be pressed before each time the user wants to start a reflex time measurement). At the same time, to handle the *STOP* button, the machine calculates and displays the average reflex time, generates a *GO_OFF* signal to turn off the light, exits from the current measurement and enters the next measurement (or finishes after three runs). Now, in the Esterel specification, we find the user annotation that the input signals *READY* and *STOP* cannot happen within the same tick (line 7). Hence our reported critical path is not a feasible one. Using the mechanism discussed in this section, such user annotations are automatically converted to (branch, branch) conflicting pair information, i.e., tests on *READY* and *STOP* cannot both be true. Naturally, this yields a tighter WCET estimate.

7 Inter-tick Micro-architectural Contexts

In this paper, we have taken advantage of the proposed program control flow analysis to obtain tighter WCET estimation and backwards traceability for performance debugging of Esterel specifications. As discussed in Section 4, micro-architecture modeling is used for accurate computation of the execution time of individual basic blocks, which is also a key component in WCET analysis, especially for general-purpose processors with complex micro-architecture. In this section, we will present a brief overview on how micro-architectural effects, the instruction cache in particular, can be integrated into our framework for a even more accurate timing analysis.

As discussed in previous sections, progress of a single Esterel tick corresponds to one complete execution of the compiled tick function. Thus, micro-architectural effects (including pipelining, cache, branch prediction, etc) can be automatically captured by the WCET analyzer for WCET calculation of the tick function. For example, [28] introduces an ILP-based WCET analysis framework integrating path analysis and instruction cache modeling, which has been extended and implemented in the Chronos WCET analyzer ([15]). As a result, the WCET values for any single Esterel tick on a given general-purpose processor (as presented in Table 2) have fully considered the intra-tick micro-architectural effects.

In a real application modeled in synchronous language, it is common for the response of an event to span across multiple clock ticks. Moreover, some micro-architecture components also make timing impacts when the tick function get executed repeatedly (corresponding to consecutive Esterel ticks), which will be referred as the inter-tick micro-architectural context in this paper. Consider the execution of the tick function for our example Esterel specification shown in Figure 2. Even though there is no inter-tick instruction reuse at the Esterel specification level, instructions in the generated C level tick function (and corresponding assembly code) may be reused across ticks. When some instructions are executed in the first tick and loaded into the instruction cache (e.g., test of the state variable s1), it is possible for them to remain in the cache and get executed again in the subsequent ticks. The inter-tick instruction cache reuses result in additional cache hits, and a smaller execution time. By considering these inter-tick contexts, accuracy of the WCET estimation can be further improved. However, such inter-tick micro-architectural contexts are not captured in the off-the-shelf WCET tools for general programs.

Timing effects of cache sharing from different program fragments have been well-studied in the literature on the cache-related preemption delay (CRPD) problem [14]. Given a preempted task T and a preempting task T', the cache-related preemption delay is an upper bound on the delay due to additional cache misses caused by preemption of T by T'. Hence, the works on CRPD analysis estimate the cache pollution (i.e., loss in execution time) due to prior execution of other program fragments.

Our problem with inter-tick cache contexts is somewhat different, but a similar cache analysis model can be adopted (e.g., the CRPD analysis presented in [33]). In particular, we want to estimate the worst-case cache reuse (i.e., the guaranteed gain in execution time) due to prior execution of the tick function. We need to compute all possible cache states at the beginning and end of the tick function. Let S be the set of all possible cache states resulted from previous tick execution, and S' be the set of cache states that contain the possible first memory references to cache blocks in the current tick. Via a pairwise comparison of the elements in S and S', we are able to locate a cache state $s_i \in S$ and a cache state $s_j \in S'$, with minimum number of common memory blocks. It gives the worst-case guaranteed cache reuse for the current tick from previous tick execution, from which the WCET of the current tick can be further tightened. A comprehensive framework that integrates the program control flow contexts and inter-tick architectural contexts will be studied in our future work.

8 Concluding Remarks

In this paper, we have presented a model-driven timing analysis framework for the single-processor execution of the Esterel specification. WCET estimation of a single Esterel tick allows validation of the synchrony hypothesis, and enables software implementation of synchronous languages on general-purpose processors. Moreover, our proposed model-driven timing analysis technique yields tighter WCET estimation, which leads to more cost-effective and resource-saving designs. By maintaining traceability between model elements (Esterel statements) and corresponding generated C (binary) instructions, the critical path reported in our WCET analysis can be mapped back to the Esterel model, which facilitates the system designer for performance debugging and further WCET refinement.

In our model-driven timing analysis, we extract information from the high-level model and Esterel-to-C compilation, which are used to provide additional program path and micro-architecture constraints to a stand-alone WCET analyzer. In particular, our analysis systematically identifies common infeasible path patterns in the generated C code from the Esterel specification. A light-weight infeasible path detection technique is proposed to automatically eliminate infeasible paths in our WCET analysis.

Comparing with our preliminary publication along this line of work ([22]), substantial improvements have been presented in this paper, including the SCFG-level infeasible path analysis, elimination of context-sensitive infeasible paths due to tick transition, and the related experimental results. In the future, we are interested in applying our model-driven timing analysis framework to other synchronous languages and extending it for the multi-core domain.

References

1. AbsInT GmbH, http://www.absint.com/.
2. C. André. Semantics of synccharts. Technical report, Sophia-Antipolis, France, April 2003.
3. T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, 2002.
4. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
5. A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.
6. R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J. P. Rigault, and J. M. Tanzi. Programming a Reflex game in Esterel v3. Technical Report 07/91, Rapport de Recherche, INRIA, Sophia-Antipolis, France, June 1991.
7. G. Berry. *Mechanized reasoning and hardware design*, chapter Esterel on hardware. Prentice-Hall, 1992.
8. V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS= Esterel+ Kronos. A tool for verifying real-time properties of embedded systems. *Proceedings of the 40th IEEE Conference on Decision and Control*, 3(4-7), 2001.
9. M. Boldt, C. Traulsen, and R. von Hanxleden. Worst Case Reaction Time Analysis of Concurrent Reactive Programs. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203(4):65–79, 2008.

10. F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 9(79):1270–1282, 1991.
11. P.Y. Chang, E. Hao, and Y.N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
12. S.A. Edwards. The Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software.html`, 2003.
13. S.A. Edwards and J. Zeng. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007.
14. C.-G. Lee *et al.* Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.
15. X. Li *et al.* Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007, `http://www.comp.nus.edu.sg/~rpembed/chronos`.
16. C. Ferdinand. *Cache behavior prediction for real-time systems*. PhD thesis, Saarland University, 1999.
17. C. Ferdinand and *et al.* Reliable and precise WCET determination for a real-life processor. In *International Conference on Embedded Software (EMSOFT)*, 2001.
18. J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.
19. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2006.
20. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.
21. R. Heckmann and *et al.* Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *4th European Congress on Embedded and Real Time Software (ERTS)*, 2008.
22. L. Ju, B.K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2008.
23. M. Kuo, R. Sinha, and P. Roop. Efficient WCRT analysis of synchronous programs using reachability. In *Design Automation Conference (DAC)*, 2011.
24. X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. Von Hanxleden. An Esterel processor with full preemption support and its worst case reaction time analysis. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005.
25. X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, 2006.
26. X. Li and R. von Hanxleden. Multi-Threaded Reactive Programming—The Kiel Esterel Processor. *IEEE Transactions on Computers*, 2010.
27. Y.T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, pages 88–98, 1995.
28. Y.T.S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, 1999.
29. G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, 2003.
30. G. Logothetis, K. Schneider, and C. Metzler. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. *Forum on Design Languages (FDL)*, 2003.
31. G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2003.
32. M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Design, Automation and Test in Europe (DATE)*, 2009.
33. H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2003.

34. D. Potop-Butucaru, R. de Simone, and J.P. Talpin. The synchronous hypothesis and synchronous languages. *The Embedded Systems Handbook*, 2004.
35. D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.
36. T. Ringler. Static worst-case execution time analysis of synchronous programs. In *5th Ada-Europe International Conference*, LNCS 1845, 2000.
37. P. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2009.
38. SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/.
39. R. K. Shyamasundar and J. V. Aghav. Realizing real-time systems from synchronous language specifications. In *RTSS Work-in-Progress Session*, 2000.
40. J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2005.
41. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference (DAC)*, 2006.
42. L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for Matlab Simulink/Stateflow. *MoDELS'09 ACES-MB Workshop*, 2009.
43. R. von Hanxleden. SyncCharts in C: a proposal for light-weight, deterministic concurrency. In *International Conference on Embedded Software (EMSOFT)*, 2009.
44. R. Wilhelm et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.
45. L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian. Compiling Esterel for distributed execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP)*, 2006.
46. S. Yuan, S. Andalam, L.H. Yoong, P.S. Roop, and Z. Salcic. Starpro–a new multithreaded direct execution platform for Esterel. *Electronic Notes in Theoretical Computer Science*, 238(1):37–55, 2009.