A Systematic Classification and Detection of Infeasible Paths for Accurate WCET Analysis of Esterel Programs

Lei Ju Bach Khoa Huynh Abhik Roychoudhury Samarjit Chakraborty

Department of Computer Science, National University of Singapore
{*julei, huynhbac, abhik, samarjit*}*@comp.nus.edu.sg*

Synchronous programming languages like Esterel are widely used in safety-critical domains like avionics. However, it is only with the recent development of mature worst-case execution time (WCET) analysis tools that progress is being made on systematically studying the WCET analysis problem for languages like Esterel. In this context, we present techniques for methodically classifying and detecting different types of infeasible paths that arise while compiling Esterel programs into executable code, via high-level languages such as C. Our experimental results with well-known benchmarks show that the infeasible paths detected using our techniques result in as much as 36.5% reduction in the WCET estimates, compared to when no infeasible path detection is employed.

1. Introduction

Synchronous languages like Esterel,[6] Lustre and Signal provide a clean formalism for programming safety-critical reactive systems that require formal verification or certification. However, due to the lack of mature software timing analysis tools for general-purpose processors, compiling such synchronous languages directly into hardware[3] – where the synchrony hypothesis is easier to validate and debug – is currently the most popular design flow.

Lately, there has been a renewed interest in timing analysis of synchronous language programs targetting general-purpose platforms (*e.g.*, see Heckmann *et al.*[10]). This is primarily due to the recent advances in worst-case execution time (WCET) analysis techniques and the availability of industry-strength WCET analysis tools (such as the aiT WCET analyzer from AbsInt GmbH[1]). Programs written in synchronous languages like Esterel are first compiled into high-level language programs such as C, which in turn are compiled for a target platform. Therefore, timing analysis of a synchronous language program necessitates the analysis of the final executable code in conjunction with the micro-architectural features of the

target platform. In this paper we conduct a systematic classification and detection of infeasible path patterns resulting from the compilation of Esterel programs. Eliminating such infeasible paths result in tighter WCET analysis of the target code, which in turn results in better resource dimensioning, performance debugging and more effective system design. While detecting infeasible paths in arbitrary C-code is a difficult problem, it turns out to be relatively simpler for code generated from Esterel specifications. Furthermore, we show that infeasible path detection and elimination can be more effectively done at an intermediate stage of the compilation process, viz., on the sequential control flow graph or SCFG (standard Esterel compilers like the Columbia Esterel Compiler[9] translate an Esterel program into C via a set of intermediate representations, one among them being the SCFG). Experimental results on a set of standard benchmarks show that our infeasible path detection techniques result in as much as 36.5% reduction in WCET estimates, compared to estimates obtained without any infeasible path detection.

The rest of this paper is organized as follows. In the next section we give an overview of the Esterel language and its compilation techniques. This is followed by a summary of WCET analysis methods in Section 3. In Section 4 we describe our proposed infeasible path detection techniques. Finally, our experimental results have been presented in Section 5.

2. Overview of Esterel

Esterel is a synchronous language where all computation and communication, unless explicitly paused (using a `pause` instruction), happen instantaneously. A run of a program 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. 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. Hence, `emit A` $\|$ `present A then emit B; pause; emit C` will emit `A` and `B` at the first tick, followed by `C` at the second tick. Further details of the syntax and semantics of Esterel may be found in[6] (or from the references in[2]).

Compiling Esterel. Various techniques exist for compiling Esterel into C programs.[13] Based on the intermediate representation used, they can be categorized into automata-based, netlist-based, and control flow graph-based

approaches. In this paper, we will focus our discussion on the control flow graph-based Esterel compilation, which normally produce fast and small C code. As mentioned before, we have integrated our work into the control flow graph-based code generation of the Columbia Esterel Compiler (CEC).[9] 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)[13] through a syntax directed translation. The GRC is then transformed into a sequential control flow graph (SCFG), via a set of intermediate representations like program dependence graph (PDG), and 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 original the Esterel program. Finally, sequential C code can be directly generated from the SCFG.

3. Overview of WCET Analysis

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 1(a) shows an example program and its control-flow graph.

Static analysis based WCET estimation proceeds by finding the longest path in the program's control flow graph, satisfying certain loop bounds (*e.g.*, in the example of Figure 1(a) the loop bound for the only loop is 10). 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.

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 control flow graph. We then maximize an objective function representing the program execution time

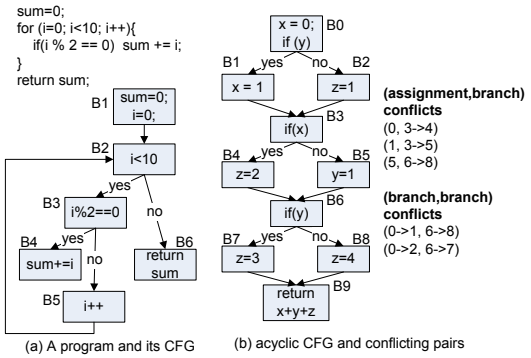

Fig. 1. Example control flow graphs.

subject to these linear constraints. Since the execution counts of control flow graph nodes/edges are integers, we can employ Integer Linear Programming (ILP) technology. Formally, let $\mathcal{B}$ be the set of basic blocks of a program. The program's WCET is given as: maximize $\sum_{B \in \mathcal{B}} N_B \times c_B$, where N_B is an ILP variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of basic block B. The linear constraints on N_B are developed from the flow equations based on the control flow graph (e.g., path information, loop bounds).

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 Figure 1(a) it is not possible for basic block 4 to execute in successive loop iterations. Thus, an infeasible path may be taken as the longest path leading to undue WCET overestimation. The lack of automation in the WCET analysis method comes from the onus on the user to provide constraints encoding such infeasible path information as well as loop bound information.

4. Infeasible Paths in Generated Program

In order to find the upper bound of the execution time for one single Esterel tick, we use a WCET analyzer to calculate the WCET of the generated C tick-function from the Esterel specification. For an architecture-aware WCET analysis, the given Esterel program is first translated into C code via a set of intermediate representations; the C code is then compiled into assembly code for WCET estimate. During the WCET estimates, infeasible paths must be excluded from the critical path (resulting the WCET) to obtain tighter results. While fully automated infeasible path detection is generally difficult for hand-written C programs, simple and effective anal-

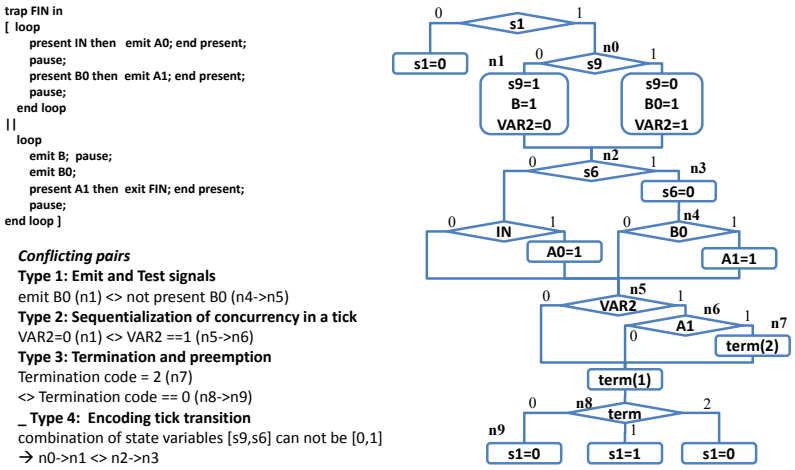

Fig. 2. Conflicting pairs in SCFG of an Esterel Program

ysis can be used for infeasible path elimination for C code generated from Esterel specification. In this work, we clarify common infeasible path patterns and show how they can be automatically detected and eliminated in WCET calculation. Furthermore, we also discuss the reason why a SCFG-level infeasible path detection is more comprehensive and efficient comparing to doing that at other higher (Esterel specification) or lower (C or binary code) levels of represention.

4.1. *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. In Columbia Esterel Compiler, C code are generated directly from the sequential CFG (SCFG).[9] Figure 2 shows an Esterel program, its SCFG (partial) and conflicting pairs between nodes/branches in the SCFG. In the SCFG, rectangle nodes represent assignments / computations, and diamond nodes represent choices. The Edge between two labeled nodes n_i and n_j are represented as $n_i \rightarrow n_j$.

We use the notion of *conflicting pairs*[14] — pairs of (assignment, branch) or (branch, branch) statements which may not appear together in an execution trace, to capture the infeasible path information. 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`.

In the SCFG (and C code) generated from Esterel, we observe the following four infeasible path patterns commonly exist.

1. Emit and test signals. The corresponding infeasible paths are also present at the SCFG level, *e.g.*, the conflicts due to assignment and test on signal $B0$ ($n4$ and $n10 \rightarrow n13$) in Figure 2. 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 variable will result in paths with (branch, branch) conflicts.

2. Sequentialization of concurrency in a tick. To produce sequential C code from a concurrent Esterel program, data dependencies and context switches between concurrent threads must be captured. In CEC, this is handled by inserting new control variables ($VAR2$ in Figure 2) and corresponding test nodes in the generated C code. Assignments and tests (may be at multiple places in the same clock tick) on the guard variable will introduce conflicting pairs.

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 ([9]). 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* in Figure 2). 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 and exception). Such assignments and the tests on these guard variables introduce possible infeasible paths.

4. Encoding tick transitions. In Esterel, ticks are executed with orders in each concurrent thread (via the use of "pause" and "await" statements). In the generated C code, these orders are 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 automata. For example, in Figure 2, given the initial value [0,0] for state variables $s6$ and $s9$, the reachable value combinations can only be [0,0] and [1,1] — which prevents the paths corresponding to [0,1] ($n6 \rightarrow n7$ and $n0 \rightarrow n2$) or [1,0] ($n6 \rightarrow n8$ and $n0 \rightarrow n1$) from getting executed within same tick.

There are many levels of intermediate representations while compiling an Esterel specification into assembly code for WCET estimate. In our work, we perform our infeasible path detection at SCFG level because of the following reasons.
1. 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 CEC translating the concurrent CFG (CCFG) into SCFG.
2. C (assembly) code level analysis is incomplete without additional instrumented code. For example, the third type of infeasible path patterns will often produce many emphswitch-cases constructs in the generated C code. the *switch* is translated into a *register indirect jump (jr)*, i.e. the branch direction can not be determined statically.
3. Improve the efficiency of conflicting pair detection. There are much less number of nodes in SCFG comparing to the number of assembly instructions. Thus, the analysis takes less time at SCFG-level.
4. The state automata (or OBDD) construction for detecting the fourth type of infeasible path is obviously easier to perform at SCFG level other than at any other lower levels (C or assembly). Furthermore, a preliminary SCFG level conflicting pair detection will make the reachable state searching more efficient and accurate.

4.2. *Conflicting Pair Detection in SCFGs*

The first three types of infeasible path patterns listed in section 4.1 are context-insensitive, i.e., the conflicts are resulted from one single execution of the SCFG (representing the tick function). They are not affected by previous executions of the program. The conflicting pairs are easy-to-compute. For (assignment, branch) conflict, we first find all pairs of conflict assignments and branches, that modify/test the same variable, and there is at least one path on the SCFG reaches the branch from the assignment node. For each such conflicting pair, we also compute the set of nodes that invalidate it, such that the invalidating node modifies the conflicting variable, and there is at least one path on the SCFG reaches the branch from the assignment node via the invalidating node. Similar process can be used to find (branch, branch) conflicts.

On the other hand, the fourth type of the infeasible path patterns is context-sensitive. The state variable's value in current tick depends on execution in previous tick. To find the infeasible paths that correspond to unreachable state variables' value combinations, we simulate the execution of SCFG to find all possible reachable system states (the initial state is

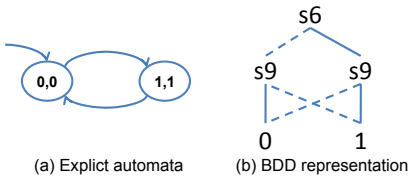

Fig. 3. Reachable state of the SCFG example in Figure 2

given). For example, Figure 3(a) shows the automata of all reachable state from the initial state ([s6=0, s9=0]) in the example SCFG in Figure 2. After constructing the automata, we generate ILP constraints for state values that can not be reached. For SCFG containing m state variables and n possible integer value for each variable, we have n^m possible states. Explicitly storing all states could be space-consuming and inefficient for large Esterel programs. An alternative is to construct an ordered binary decision diagram (OBDD)[7] on-the-fly, which can dramatically reduce the space. For each CNF clause corresponding to a path in the OBDD ends at leave node 0, an infeasible path constraint is generated. Figure 3(b) shows the OBDD representation of the reachable states in the example SCFG in Figure 2.

A conflicting pair in SCFG captures a pair of statements which cannot be executed together in one single execution of the tick function. Similarly as for the assembly level conflicting pairs discussed in Ju *et al.*,[11] such a conflicting pair is valid only if *the variable resulting in the conflict is not modified in between the execution of these two statements*. Let N_i, $E_{j\to k}$ be the execution count of node i and edge $j \to k$ in a SCFG, respectively; and $invalid(i, j \to k)$ ($invalid(i \to l, j \to k)$) be the set of node who may invalid the conflict between node i (edge $i \to l$) and edge $j \to k$. In particular,

$$invalid(i, j \to k) = \{p | reach(i, p) \wedge reach(p, k) \wedge assign(p, \mathtt{x})\}$$

where $reach(i, p)$ is true if there is a path from basic block i to basic block p, $reach(p, k)$ is true if there is a path from basic block p to basic block k, and $assign(p, \mathtt{x})$ is true if $\mathtt{x}$ is modified in the basic block B_p. Thus, if any basic block in the set $invalid(i, j \to k)$ is executed, the conflict between the assignment on $\mathtt{x}$ in block i and the test on $\mathtt{x}$ in edge $j \to k$ is no longer valid (simply because variable $\mathtt{x}$ gets modified).

ILP constraints to capture the infeasible path information encoded by (assignment, branch) and (branch, branch) conflicting pairs are then defined as $N_i + E_{j\to k} - \sum_{p\in invalid(i,j\to k)} N_p \leq 1$, and $E_{i\to j} + E_{k\to l} - \sum_{p\in invalid(i\to j,k\to l)} N_p \leq 1$, respectively. Since each node in SCFG contains only one atomic operation (assignment or test on a variable), it will be mapped to a single basic block in the C/assembly code. Above ILP constraints can be easily converted into a ILP constraints at basic block level, and fed into an ILP-based WCET analyzer.

5. Experimental Results

In this section, we show our experimental results by comparing the WCET estimates for Esterel specifications with and without the infeasible path detection. Esterel programs were compiled into C using the default code generation mechanism (GRC-based) in the Columbia Esterel Compiler (CEC).[9] We instrumented CEC by adding comments in the generated C code so that during the compilation a C-Esterel mapping is created. We used Chronos,[12] an ILP-based WCET analyzer, to calculate the WCET of the tick function in the generated C code. For the WCET analysis, we assume the following architectural configuration: a direct mapped L1 instruction cache, perfect branch predication, 5-staged pipeline, and an instruction dispatch queue size of 4. We used benchmarks from Estbench Esterel Benchmark Suite.[8]

Table 1. WCET analysis results.

Benchmark	# of Esterel lines	# of C lines	WCET (cycles)		reduction
			w/o inf.	w/ inf.	
runner	55	253	2814	2568	8.7%
reflex	96	378	3974	3471	12.7 %
abcd	101	827	8467	7358	13.1%
wristwatch	1088	1755	22851	14518	36.5%

Table 1 summarizes the results we obtained. For each program, we show the code size of the Esterel specification and the generated C program. We also show the WCET estimates with and without the SCFG-level infeasible path detection for each benchmark. For moderately-sized programs like the "wristwatch" – which contains many concurrent processes – the WCET result is improved by more than 36% percent by eliminating infeasible paths from the program's critical path.

6. Related Work and Concluding Remarks

High-level timing analysis of Esterel programs have been studied before,[4] where the problem was to compute the number of transitions in the underlying automata (encoding an Esterel specification) in response to different input events. Low level WCET analysis for a single Esterel tick is solved in Boldt *et al.*[5] for a special Esterel processor, where the instruction set and micro-architecture are different from a general-purpose processor. Our work on the other hand, systematically studies the infeasible path detection problem for Esterel programs.

In future, we will evaluate our techniques on larger case studies. It is also insteresting to compare our techinque with similarly approaches that validating synchrony hypothesis on general-purpose processors by perform-

ing separated WCET analysis. For example in the combination of SCADE scuite and aiT tool, the WCET anlyzer might not be awared of the fact that the anlayzed C code is generated from high-level synchronous specifications, but performa a general-case infeasible path elimination. Furthmore, we also plan to apply our framework to other synchronous programming environments (e.g., SCADE 6) and differnt code compilation techinques.

Acknowledgments

This work was partially supported by NUS URC grant R252-000-321-112.

References

1. AbsInT GmbH, http://www.absint.com/.
2. 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.
3. G. Berry. *Mechanized reasoning and hardware design (Chapter in Esterel on hardware)*. Prentice-Hall, 1992.
4. 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.
5. 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.
6. F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 9(79):1270–1282, 1991.
7. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. S.A. Edwards. The Estbench Esterel Benchmark Suite. *http://www1.cs.columbia.edu/ sedwards/software.html*, 2003.
9. S.A. Edwards and J. Zeng. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007.
10. R. Heckmann *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.
11. L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *ACM International Conference on Hardware Software Codesign and System Synthesis (CODES+ISSS)*, 2008.
12. X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007.
13. D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.
14. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, 2006.