Performance Debugging of Esterel Specifications

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

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", where the computation/communication associated with the processing of all events occurring within the same "clock tick" are assumed to happen instantaneously (or in zero time). In reality, Esterel specifications get compiled to implementations (such as C code) which do not satisfy the perfect synchrony assumption. Hence, platform-specific timing analysis of such implementations is an important research topic. Interest in this area has lately been renewed with the recent advances in Worst-case Execution Time (WCET) analysis techniques. In this paper we perform WCET analysis on sequential C code and exploit the structure of the code generated from Esterel specifications to obtain tight WCET estimates. Such estimates can validate Esterel-level assumptions on the instantaneous processing of signals or events that occur together. More importantly, they can be used to identify parts of the specification which might pose as timing/performance bottlenecks with respect to the underlying platform. This is done by exploiting traceability links between Esterel specifications and the generated C code, which map the time-critical computations at the C-level back to the Esterel-level. This not only allows a designer to optimize or simplify Esterel specifications, but also choose/configure suitable implementation platforms. We show the results of our WCET analysis on a set of standard Esterel benchmarks and illustrate the utility of our model-code traceability technique using an Esterel specification of a reflex game application.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms

Design, Languages, Performance

Keywords

Esterel, Synchronous programming, WCET analysis

1. INTRODUCTION

For safety-critical domains, synchronous programming [1] 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 [7], Lustre [11] and Signal [2] – 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. In reality, specifications in synchronous languages are compiled to implementations in high-level languages such as C, which in turn are compiled for a target platform. Such implementations clearly do not follow the *perfect synchrony* hypothesis and the time associated with different computation/communication tasks depend on the implementation platform. However, a real-life implementation 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. Hence, 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).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-470-6/08/10 ...$5.00.

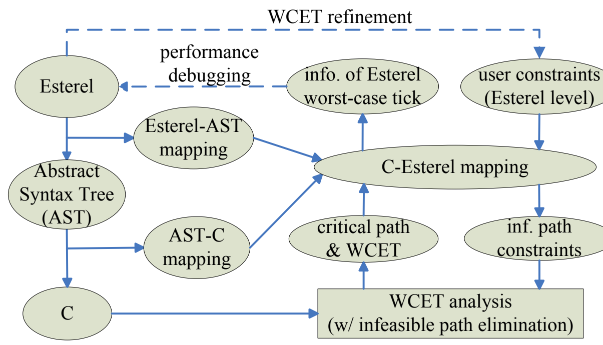

WCET refinement
performance debugging
Esterel
info. of Esterel worst-case tick
user constraints (Esterel level)
Esterel-AST mapping
C-Esterel mapping
Abstract Syntax Tree (AST)
AST-C mapping
critical path & WCET
inf. path constraints
C
WCET analysis (w/ infeasible path elimination)

Figure 1: WCET analysis framework for Esterel specifications.

Currently, a systematic design process is missing in the context of synchronous languages when the target platform is a general-purpose processor. As a result, compiling synchronous language specifications directly into hardware [4] – where the synchrony hypothesis is easier to validate and debug – is currently the most popular design flow. This has primarily been due to the lack of mature software timing analysis techniques for general-purpose processor architectures. However, recent advances in Worst-case Execution Time (WCET) analysis techniques and the availability of industry-strength tools (see [18]) has renewed the interest in synchronous language-based design flows targeting general-purpose platforms (*e.g.*, see [10]).

In this paper we propose WCET analysis techniques – which exploit the special structure of the C-code generated from Esterel specifications – to obtain tight estimates on the WCET of computations that are logically assumed take zero time. Not only can this help to validate the synchrony hypothesis without introducing undue pessimism in the WCET estimates, our technique can also be used to feed the results of the analysis back to the specification using code-model traceability links. Such feedback can be used to identify potential bottlenecks in the specification, which can then be used to debug the specification or configure the implementation platform. An overview of our methodology appears in Figure 1.

Our Contributions. Our main technical contributions include (i) light-weight techniques for systematically identifying and removing infeasible program paths (paths which do not appear in the execution trace for any program input) in the C-code generated from Esterel, thereby leading to tighter WCET estimates, and (ii) techniques for bidirectional traceability between the executable code (obtained from compiling the generated C code for a specific instruction set architecture) and the Esterel specification. For both these tasks we have used the well-known Columbia Esterel Compiler (CEC) [9]. However, our proposed techniques are very general in nature. The main novelty of our work 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. This simplifies the problem to a large extent, especially when compared to arbitrary C code for which the infeasible path detection problem is intractable. Second, WCET analysis for arbitrary programs is not (and can never be) a fully automated process. It requires substantial user intervention in the form of providing loop-bounds and infeasible path annotations. However, the solution to the WCET analysis problem in the context of Esterel is very close to a fully automated one; we discuss the automation issues in Section 4. We have implemented the framework shown in Figure 1 by integrating the Columbia Esterel Compiler with the Chronos WCET analyzer [12] that allows detailed processor modeling (*e.g.*, cache, in-order and out-of-order pipeline, branch prediction).

Related Work. There is an existing body of work which seek to use new special-purpose processors for executing specifications written in synchronous languages like Esterel (*e.g.*, see [6]). We believe that in the real world, specifications written in modeling languages like Esterel will get compiled to C/Java code and get executed on off-the-shelf processors. Thus, it is meaningful to develop/deploy software timing analysis methods for this purpose.

Analysis and debugging of timing properties of synchronous language specifications (targeting general-purpose processors) has been somewhat ignored until recently. High-level timing analysis of Esterel programs have been studied in [5, 16], where the problem was to compute the number of transitions in the underlying automata (encoding an Esterel specification) in response to different input events. In other words, the *high-level* timing 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 [13].

Low level WCET analysis for a single Esterel tick is solved in [6] for a special Esterel processor, where the instruction set and micro-architecture are different from a general-purpose processor. The problem addressed in [15] is the closest to what we study in this paper. 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. Further, 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. Finally, very recently, [10] reports preliminary results and plans to integrate an industry-strength WCET analyzer with the SCADE tool-suite from Esterel Technologies. The framework we propose here follows a similar line of work and to the best of our knowledge is the first systematic study on performance debugging of Esterel specifications targeting general-purpose processor architectures.

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. Hence, 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. 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 [7] (or from the references in [1]).

Compiling Esterel. Various techniques exist for compiling Esterel into C programs [14]. 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) [14] 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 2(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 2(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,

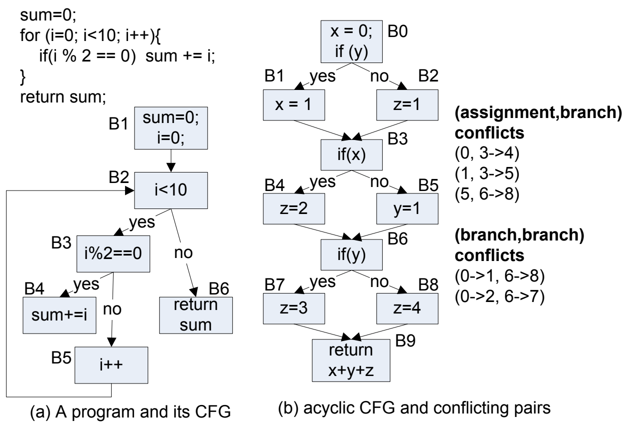


Figure 2: **Example control flow graphs.**

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 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:

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

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. 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 2(a) we need to add the constraint $E_{5 \to 2} \leq 10$).

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 2(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. ANALYSIS OF GENERATED CODE

We compile a given Esterel program into C and calculate the WCET of the C code via a platform-aware WCET analyzer. Fortunately, for C code generated from Esterel specifications, the user can largely avoid the problems with precision and automation of the WCET analysis. In this case, we analyze a C function called the *tick-function* which encodes all possible computations within a clock tick allowed by the Esterel specification. Since the tick-function is loop-free (Esterel allows no loops within a clock tick), this leads to an acyclic CFG and hence there is no need to provide loop bounds to the WCET analyzer. Thus, each basic block is executed at most once, and ILP-based WCET analysis produces a 0-1 assignment for the execution count of each basic block. Furthermore, we use the structure of the generated code to efficiently detect/exploit common infeasible path patterns (leading to tighter WCET estimates).

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. Central to our approach is the notion of *conflicting pairs* [17] — 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. Figure 2(b) shows an example acyclic CFG and certain infeasible path patterns in it. We show examples of (assignment, branch) as well as (branch, branch) conflicting pairs.

The notion of conflicting pairs is simple and easy-to-compute. Clearly, it will not allow us to detect infeasible paths of the form `x=5; z=x; if (z > 5)...` However, as we see in the following, in automatically generated C code from Esterel — the notion of conflicting pairs captures many common infeasible path patterns. Based on our study of C programs generated via Esterel compilation to sequential control flow graphs, we found the following four common sources of infeasible paths. For each of these four sources, in Figure 3 we show example Esterel program fragments (labeled with line numbers) and the corresponding C code (labeled with basic block numbers) generated by the default code generation mechanism in the Columbia Esterel Compiler [9]. The four infeasible path pattern types are as follows.

1. Emit and test signals. The corresponding infeasible paths are also present at the C level, *e.g.*, the conflicts due to assignment and test on signal A ($B1$ and $B2 \to B4$) in the first program fragment in Figure 3. 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 generate 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 and corresponding test nodes in the generated C code. In the first program fragment (Figure 3), the variable $\_DPSCUT\_VAR2$ captures the *state* of the first thread before a context switch, and is used as a conditional guard when the thread resumes execution. Such 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

Example1 (Type 1 & 2)		Example2 (Type 3)		Example3 (Type 4)	
Esterel	Generated C code	Esterel	Generated C code	Esterel	Generated C code
L1: emit A;	B1: A = 1;	…	…	L1: loop	B1: if (_state_3) {
L2: present B then	B1: _DPSCUT_VAR2 = 0;	L1: trap T in	B1: A = 1;	L2: emit A0;	B2: A0 = 1; _state_3 = 0;
L3: emit C;	…	[	B1: _term_17 &= -(1 << 2); //exit T	L3: pause;	}
L4: end present	B2: if (A)	L2: emit A;	B1: B = 1;	L4: emit A1;	else {
…	B3: B = 1;	L3: exit T;	B1: _term_17 &= -(1 << 1); //pause	L5: pause;	B3: A1 = 1; _state_3 =
L5: ||	else	…	…	L6: end	1;
…	B4: D = 1;	L4: ||	B2: switch (~_term_17) {	L7: ||	}
L6: present A then	…	L5: emit B;	B3: case 0: … break;	L8: loop	B4: if (_state_6) {
L7: emit B;	B5: if (_DPSCUT_VAR2)	L6: pause;	B4: case 1: … //pause	L9: emit B0;	B5: B0 = 1; _state_6 = 0;
L8: else	{ … }	…	B4: break;	L10: pause;	}
L9: emit D;	else {	]	B5: case 3: … C = 1; //exit T	L11: emit B1;	else {
L10: end present	B6: if (B) C = 1;	L7: emit C	B5: break;	L12: pause;	B6: B1 = 1; _state_6 = 1;
	}		…	L13: end	}

Figure 3: Example infeasible path patterns in generated C code.

([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 $\_term\_17$ as in the second example in Figure 3). 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, a global automata 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 automata 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 automata. For example, in the third program fragment, given the initial value [0,0], the combinations of values to ([$\_state\_3$, $\_state\_6$]) can only be [0,0] or [1,1] — which prevents the paths corresponding to assignments [0,1] or [1,0] in getting executed. We found that this kind of infeasible paths may *not* necessarily be captured via conflicting pairs.

4.2 Infeasible Path Elimination

We now discuss methods to detect/exploit conflicting pairs in the ILP-based WCET analysis of the acyclic tick-function code generated from Esterel. First, we compute all conflicting pairs of (assignment, branch) and (branch, branch) statements in the acyclic control flow graph of the generated tick-function code. This is done in $O((|N| + |E|) * |E|)$ time where $|N|$ and $|E|$ are the number of nodes and edges in the control flow graph of the tick function. For every branch edge we may need to test (i) all other branch edges, and (ii) all other nodes containing assignment statements.

Even after the conflicting pairs are detected, we cannot directly use them in our ILP-based WCET analysis. Suppose we find that an assignment to a variable $\mathtt{x}$ in block i conflicts with a branch edge $j \rightarrow k$ (edge between basic block j and basic block k) on variable $\mathtt{x}$. A straightforward encoding of this conflicting pair as a linear constraint would be $N_i + E_{j\rightarrow k} \leq 1$ where N_i ($E_{j\rightarrow k}$) is the 0-1 execution count of block i (edge $j \rightarrow k$). The above constraint means that block i and edge $j \rightarrow k$ cannot be executed together. However, 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*. For example, in Figure 2(b), the assignment to variable $\mathtt{x}$ in block 0 conflicts with the test of $\mathtt{x}$ along the branch edge $3 \rightarrow 4$. However, this (assignment, branch) pair is conflicting (*i.e.*, it cannot appear together in an execution trace) only if $\mathtt{x}$ is not modified in between the execution of block 0 and edge $3 \rightarrow 4$. In other words, for this conflicting pair to be valid, basic block 1 (which modifies the value of $\mathtt{x}$) must not be executed in between. This leads to the constraint

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

Formally, we can encode conflicting pairs as linear constraints on the 0-1 execution counts of nodes/edges in the control flow graph. Consider an (assignment, branch) conflicting pair involving an assignment to a variable $\mathtt{x}$ appearing in basic block i and a branch edge $j \rightarrow k$ testing the same variable $\mathtt{x}$. We first define the following quantity

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

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 $invalid(i, j \rightarrow k)$ is executed, the conflict between the assignment on $\mathtt{x}$ in block i and the test on $\mathtt{x}$ in edge $j \rightarrow k$ is no longer valid (simply because variable $\mathtt{x}$ gets modified). This can be encoded as the linear constraint

$$N_i + E_{j\rightarrow k} - \sum_{p \in invalid(i,j\rightarrow k)} N_p \leq 1$$

where N_i (N_p) is the execution count of basic block i (p) and$E_{j\rightarrow k}$ is the execution count of edge $j \rightarrow k$.

For (branch, branch) conflicting pairs, we also define a similar quantity $invalid(i \rightarrow j, k \rightarrow l)$, where $(i \rightarrow j, k \rightarrow l)$ is a conflicting pair of branch edges on a variable $\mathtt{x}$, and $invalid(i \rightarrow j, k \rightarrow l)$ captures all basic blocks modifying $\mathtt{x}$, and appearing in a path from $i \rightarrow j$ to $k \rightarrow l$. We can then encode this conflicting pair as a linear constraint

$$E_{i\rightarrow j} + E_{k\rightarrow l} - \sum_{p \in invalid(i\rightarrow j,k\rightarrow l)} N_p \leq 1$$

Note that we detect all the conflicting pairs *automatically* and corresponding to each such pair, the above linear constraint is *automatically* added to the ILP formulation of the WCET analysis problem (thereby leading to a tighter WCET estimate). Among the four infeasible path pattern types discussed in Section 4.1, the first three patterns of infeasible paths can be easily detected using (minor extensions of) our conflicting pair-based infeasible path detection technique. However, the last type of infeasible paths – caused by the encoding of state variables – is difficult to identify. To handle this kind of paths, we allow the programmer to provide infeasible path annotations at the Esterel level and automatically translate them into basic block level linear constraints to refine the WCET estimate. This is elaborated in the next section.

5. BACKWARDS TRACEABILITY

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 (Figure 1). This mapping is

Benchmark	# of Esterel lines	# of C lines	# of conflicting pairs		runtime	WCET (cycles)		reduction
			(assgn, branch)	(branch, branch)		w/o inf.	w/ inf.	
runner	55	253	22	20	7.2s	2996	2724	9.1%
reflex	96	378	65	40	12.4s	4776	3910	18.1%
abcd	101	827	527	1296	27.2s	9881	7955	19.5%
mejia	555	2598	1225	4737	1m57s	18257	15983	12.5%
wristwatch	1088	1755	806	4869	2m26s	23320	17271	25.9%
mca200	7269	10894	845	2873	15m25s	63660	52016	18.3%

Table 1: WCET analysis results.

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.

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.

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; this is done only for the Esterel statements we want to trace. Subsequently the AST is transformed into C via changes in control flow due to sequentialization of Esterel's concurrency. However, the computation/predicate nodes of the AST that we trace are retained in the C control flow graph. Hence, we can map the AST nodes to statements in the C control flow graph. By composing the Esterel to AST and the AST to C mappings, we get a mapping from Esterel program line numbers to C program line numbers.

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.

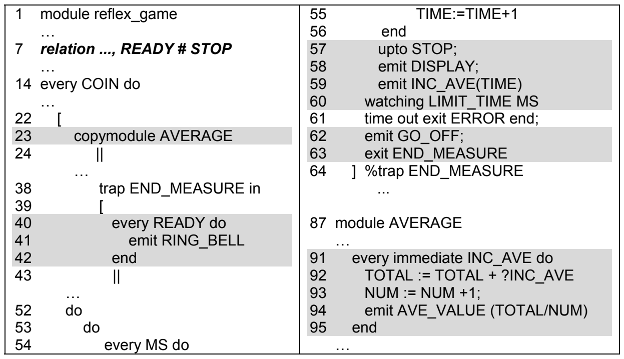
What kind of infeasible path annotations can be provided at the Esterel level? Esterel allows the programmer to explicitly define # (exclusion) and => (implication) relations on signals. These are constraints on the environment of the Esterel specification (*e.g.*, signals x and y never happen in the 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 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. EXPERIMENTS

In this section, we present some implementation details and experimental results to evaluate our proposed analysis framework. We compiled Esterel programs into C using the default code generation mechanism in the Columbia Esterel Compiler (CEC) [9]. We instrumented CEC 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, the default architectural configuration of the tool was used, which assumes a direct mapped L1 instruction cache, dynamic 2-level branch predictor, 5-staged pipeline, and an instruction dispatch queue size of 4. For infeasible path detection, we implemented the infeasible path detection method as discussed in Section 4.1, which automatically detects (assignment, branch) and (branch, branch) conflicting-pair information from the program's CFG. The generated constraints were provided to the WCET analyzer for tighter WCET analysis.

We used benchmarks from Estbench Esterel Benchmark Suite [8], including a runner's behavioral description (runner), a simple combination lock (abcd) and the well-known Wristwatch example from the Esterel distribution. We also studied the reflex game example [3], which we discuss in details as a case study.

Table 1 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 the infeasible path detection algorithm are also listed. The *runtime* column shows the running time of the entire WCET analysis, including infeasible path detection, ILP constraint generation, and ILP solving. Finally, the calculated WCET values with and without the infeasible path detection for each benchmark


```
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
    …
```

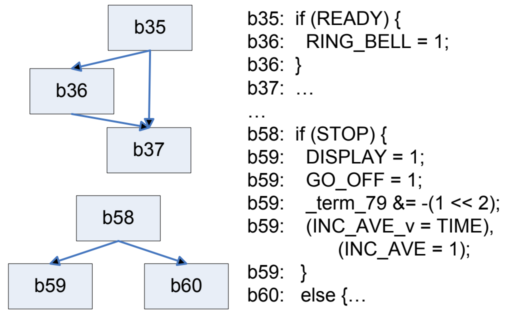
Figure 4: Esterel-level critical path of the reflex game

is presented. We can see a WCET reduction varying from 9.1% up to 25.9% resulting from infeasible path detection. A tighter WCET value improves the accuracy of the synchrony hypothesis validation, and provides system engineers with more flexibility in term of design choices. Moreover, with infeasible path detection in-built into the WCET analysis, we can construct a feasible critical path in C — which can then be mapped back to the Esterel level.

Case Study. We illustrate our timing analysis framework using the well-known reflex game example [3]. 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 4 shows an Esterel fragment of the game controller. The complete Esterel specification of the game can be found in [3] (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 (Sec. 4) 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 (Sec. 5). Figure 5 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 5 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 5 corresponds to Esterel lines 40-42, 58-63 in Figure 4.

The entire Esterel level critical path is shown using shaded lines in Figure 4. 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 Section 5, 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.



```
b35:  if (READY) {
b36:    RING_BELL = 1;
b36:  }
b37:  …
…
b58:  if (STOP) {
b59:    DISPLAY = 1;
b59:    GO_OFF = 1;
b59:    _term_79 &= -(1 << 2);
b59:    (INC_AVE_v = TIME),
          (INC_AVE = 1);
b59:  }
b60:  else {…
```

Figure 5: C-level critical path of the reflex game

7. CONCLUDING REMARKS

In this paper, we have used WCET analysis of generated C code from Esterel specifications to validate the synchrony hypothesis. If the maximum computation in a clock tick is found to exceed the clock tick period, we map the longest or critical path in the C code back to Esterel for performance debugging/optimization of the specification. We are currently in the process of applying our analysis/debugging methodology to other synchronous languages.

Acknowledgments

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

8. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] G. Berry. *Mechanized reasoning and hardware design*, chapter Esterel on hardware. Prentice-Hall, 1992.

[5] 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.

[6] 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.

[7] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 9(79):1270–1282, 1991.

[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] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

[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] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *RTSS*, 2003.

[14] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.

[15] T. Ringler. Static worst-case execution time analysis of synchronous programs. In *5th Ada-Europe International Conference*, LNCS 1845, 2000.

[16] R. K. Shyamasundar and J. V. Aghav. Realizing real-time systems from synchronous language specifications. In *RTSS Work-in-Progress Session*, 2000.

[17] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, 2006.

[18] 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.