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).

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,
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 on a variable 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)... 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)... 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.
an assignment to a variable use them in our ILP-based WCET analysis. Suppose we find that and (ii) all other nodes containing assignment statements.

every branch edge we may need to test (i) all other branch edges, nodes and edges in the control flow graph of the tick function. For

equation would be

$$x \in \{0,1\}$$ or \(\{1,0\}\) in getting executed. We found that this kind of invalid paths may not necessarily be captured via conflicting pairs.

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 \(\mathcal{O}(|N| + |E| + |E|)\) 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 \(x\) in block \(i\) conflicts with a branch edge \(j \rightarrow k\) (edge between basic block \(j\) and basic block \(k\)) on variable \(x\). A straightforward encoding of this conflicting pair as a linear constraint would be \(N_i + E_{j \rightarrow k} - \sum_{p \in invalid(i,j \rightarrow k)} N_p \leq 1\) where \(N_i\) is the execution count of basic block \(i\) 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 \(x\), and \(invalid(i \rightarrow j, k \rightarrow l)\) captures all basic blocks modifying \(x\), 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...
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.

We compiled Esterel programs into C using the default code generator 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.
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 terms 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 after 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 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.

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