Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs

Sidharta Andalam and Partha S Roop
Department of Electrical and Computer Engineering
University of Auckland, New Zealand
Email: {sand080, p.roop}@aucklanduni.ac.nz

Alain Girault
INRIA Grenoble Rhône-Alpes
LIG, POP ART team, Grenoble, France
Email: Alain.Girault@inria.fr

***Abstract*—Synchronous programs execute in discrete instants, called ticks. For real-time implementations, it is important to statically determine the worst case tick length, also known as the worst case reaction time (WCRT). While there is a considerable body of work on the timing analysis of procedural programs, such analysis for synchronous programs has received less attention. Current state-of-the art analyses for synchronous programs use integer linear programming (ILP) combined with path pruning techniques to achieve tight results. These approaches first convert a concurrent synchronous program into a sequential program. ILP constraints are then derived from this sequential program to compute the longest tick length.**

In this paper, we use an alternative approach based on model checking. Unlike conventional programs, synchronous programs are concurrent and state-space oriented, making them ideal for model checking based analysis. We propose an analysis of the abstracted state-space of the program, which is combined with expressive data-flow information, to facilitate effective path pruning. We demonstrate through extensive experimentation that the proposed approach is both scalable and about 67% tighter compared to the existing approaches.

I. Introduction

Synchronous programs have a formal model of computation based on the synchrony hypothesis. This facilitates formal analysis and also ensures that all *correct* synchronous programs satisfy determinism (safety) and reactivity (liveness) properties. This hypothesis abstracts time into discrete instants so that outputs happen instantaneously relative to the inputs. In a nutshell, this hypothesis requires that the *idealized synchronous program executes infinitely fast relative to its environment*. For all practical purposes, however, the synchrony hypothesis can be respected if the maximum length of computation within an instant is less than the minimum inter-arrival time of events from the environment, i.e., the system is *fast enough* compared to its environment. Hence, it is important to be able to statically determine the worst case execution time of any instant (also known as a tick) in order to determine whether the synchrony hypothesis can be respected for a given environment. Also, to guarantee timing repeatability for real-time systems, it is essential to fix the tick length to this worst case tick length, also known as the worst case reaction time (WCRT) [6] of the system.

Unlike WCRT analysis that computes the longest tick of a synchronous program, worst case execution time (WCET) analysis is the process of determining the worst cases execution path in a given program [19]. While a plethora of techniques exist for WCET analysis of procedural programs, there are only a handful of techniques for determining the WCRT of synchronous programs.

For synchronous programs, the timing analysis needs to consider the following issues. Synchronous programs are logically concurrent, have complex control flow that can be preempted, and have state-boundaries in every thread (called *local ticks*) that must be synchronized using some form of *barrier synchronization* to determine global ticks. Early work on the timing analysis of synchronous programs focused on the conventional max-plus based approaches [6], where the global tick length is computed by summing up all the maximum local tick lengths. Subsequently, ILP-based formulation has been presented in [11] where a synchronous program is first converted to sequential C code using the CEC [8] compiler for Esterel. ILP constraints are then derived to compute the longest tick, while also taking infeasible paths in the resulting C program into account. More recently, the same researchers noticed that synchronous programs have both variable-value based infeasible paths and state-based infeasible state combinations. Hence, they have refined their ILP formulation further to prune redundant states by imposing a further execution automation to the sequential C code [10]. Our opinion is that, to compute a tight WCRT value, it is counter-intuitive to convert a concurrent program to a sequential program before superimposing additional state information for pruning of infeasible paths.

Synchronous programs, being concurrent and state-space oriented, are ideal for model checking based analysis [16], where such analysis can be done by extracting information from a concurrent intermediate format directly. Earlier model checking based analysis for WCET [16] were not based on an abstraction of the program. We demonstrate that abstraction-based model checking can be scalable as well as tight. This is feasible because model checking facilitates not only effective modeling of concurrency and state-space exploration, but also techniques for computing loop bounds and infeasible path pruning. Earlier works on model checking based analysis were of exponential complexity, since either timed automaton with real-valued clocks were used [5] or synchronous Kripke structures were created [14]. Also, algebraic formulation of

the same problem [15] was also recently developed, where infeasible path pruning did not include state dependencies. The paper advances the state-of-the-art in the following ways:
(1) We propose the first model checking based efficient formulation for WCRT analysis of synchronous program that combines abstraction-based model checking with very efficient techniques for pruning infeasible paths. (2) The proposed method works directly on the concurrent program description. This approach is much more natural and scalable compared to the ILP formulation on the sequential equivalent of a concurrent program. (3) Through experimentation, we demonstrate how tightness can be improved thanks to the pruning of infeasible paths, while at the same time improving scalability of model checking.

II. Overview of PRET-C

For our analysis, we have selected a synchronous variant of C called PRET-C [2], which is similar in spirit with an earlier synchronous C variant called ReactiveC [7]. We selected PRET-C over Esterel [4] thanks to its support for light-weight multi-threading, causality by construction, and support for shared variables in C, making it an ideal language to design embedded systems. Compared to ReactiveC, PRET-C is essentially C thanks to its macro-based implementation, and supports a slightly different notion of thread priorities to ensure causality [3]. Yet, the proposed methodology of WCRT analysis developed for PRET-C is generic and can be also applied to the analysis of Esterel and ReactiveC.

In PRET-C, threads communicate through global variables. The proposed semantics ensures that shared memory access is thread-safe by construction [2]. PRET-C supports strong and weak preemptions that are similar to immediate preemptions in Esterel. PRET-C threads have fixed priority and are compiled to a single function where "multithreading" is elicited through context switching by using a barrier synchronisation statement called `EOT`. Each `EOT` marks the end of the *local tick* of its thread. Concurrent threads are launched with the `PAR` construct. A *global tick* elapses only when all participating threads of a `PAR` reach their respective `EOT`. In this sense, `EOT` is similar to the `pause` statement of Esterel, enforcing synchronization between threads.

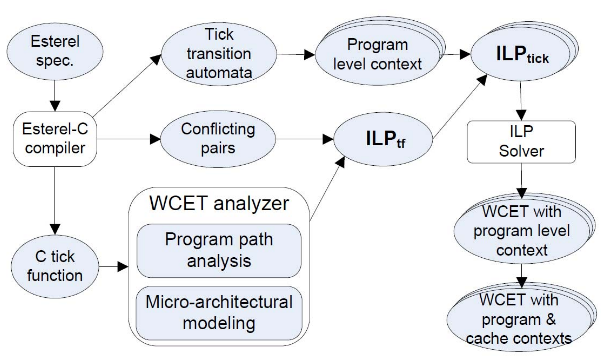
Fig. 1 presents a PRET-C example. PRET-C macros are included from the `pretc.h` file. Line 2 declares a reactive input `rst`, which is sampled at the start of each tick. Line 3 declares a reactive output `out`, which is emitted to the environment at the end of the tick. Line 4 declares a global variable `j` for shared communication between threads `T1` and `T2`. Thread `T1` executes for three ticks before terminating, while thread `T2` has a loop, and depending on the value of `j`, can execute for more than two ticks. The main thread spawns `T1` and `T2` using the `PAR` statement on line 36. The textual ordering gives `T1` priority over `T2`. This PAR is nested inside a strong `Abort` which evaluates the preemption condition `rst==j` (line 37) at the beginning of every tick. If the `PAR` is preempted, both threads terminate. Then, the reactive output `out` is updated with the value of `j`. More details of the PRET-C language appear in [2].

```
 1 #include <pretc.h>
 2 ReactiveInput(int,rst,0);
 3 ReactiveOutput(int,out,0);
 4 int j=0, x,y,z;
 5 void T1(){
 6   x=2;
 7   if(x<z)
 8     j=2;
 9   else
10     j=3;
11   EOT;
12   y=z-x;
13   if(y){
14     j=20;
15     EOT;
16   }else
17     j=35;
18     EOT;
19   }
20   EOT;
21 }
22 void T2(){
23   int i;
24   EOT;
25   for(i=0;i<j;i++)
26   {
27     j--;
28     EOT;
29   }
30   EOT;
31 }
32 int main(){
33  while(1){
34  z=3;
35  Abort{
36    PAR(T1,T2);
37  }when(rst==j);
38  out=j;
39  EOT;
40  }
41 }
```

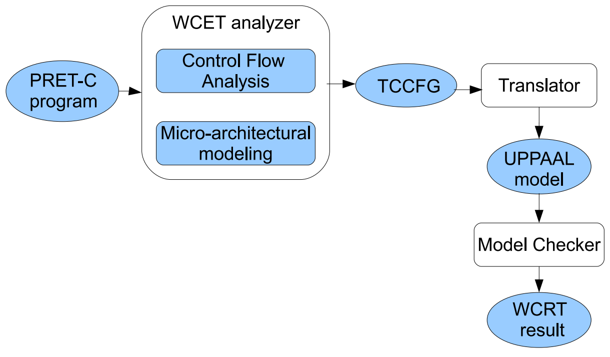
Fig. 1. Running example to illustrate various infeasible paths

III. Static timing analysis

State-of-the-art timing analysis for synchronous programs is based on ILP. Fig. 2 presents a comparative overview of the ILP-based timing analysis approach [10], [12] and our Model Checking based approach. The ILP approach (Fig. 2a) for timing analysis of Esterel relies on the CEC compiler [8] to translate Esterel into sequential C code. The generated C code is then compiled into assembly. Then, the WCET analyzer extracts the temporal properties. The control information is obtained from the compiler's intermediate format Sequential Control Flow Graph (SCFG). Then, a set of ILP constraints are generated. Finally, an ILP solver is used to compute the WCRT of the program. Our approach (Fig. 2b) takes as input a PRET-C program. Unlike the ILP-based approach, we do not compile away the concurrency. We first compile the PRET-C program using the gcc compiler to obtain the assembly code for the target processor (with compiler optimizations switched off). From the assembly code, we then extract the concurrent control-flow of the program along with its temporal characteristics, which are obtained through a hardware model of the processor (captured by a control-flow graph with transition costs, called the timed concurrent control flow graph, TCCFG). Fig. 3 shows the TCCFG of our running example. Each node of the TCCFG is annotated with the number of clock cycles that are required to execute it e.g., the "`j=3`" block of `T1` requires two clock cycles. EOT is implemented as a macro, which invokes the scheduler for context switching; this requires 17 clock cycles. Also, processor does not use any branch prediction: every conditional node's false branch has an extra cost of five clock cycles to account for pipeline flushing (see the "`j--`" block of thread `T2`, where we have +5 to indicate the cost of pipeline flush). More details are presented in [2]. As illustrated in the next section, TCCFG is an ideal input format for an abstraction-based model checking



(a) Integer Linear Programming (taken from [10]).



(b) Model Checking (this paper).

Fig. 2. Framework comparison between ILP based approach and Model Checking approach.

solution.

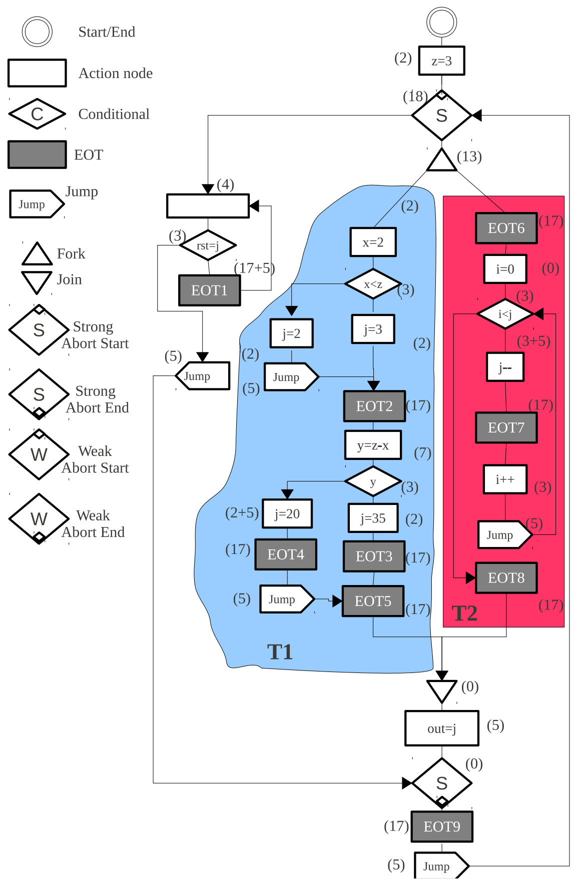


Fig. 3. TCCFG of the running example

A. Model Checking Formulation

We convert a TCCFG into a set of equivalent timed automata (TA), where execution costs is captured on the transitions between the nodes. We use the UPPAAL model checker [1] to model our automata. We do not use any clocks but only one bounded integer to capture the execution cost. This differs from earlier work on the analysis of synchronous programs, where timed automata (TA) with real-valued clocks were used [5], for which the complexity has been shown to be PSPACE-complete [13]. Our choice of UPPAAL is mainly due to our familiarity with this tool, combined with its excellent user interface. Still, the proposed approach is applicable to any other model checker with support for bounded integer counters.

Fig. 4 presents an automaton that captures a very abstracted model of the running example. In this abstracted model, we only capture the tick boundaries of each thread. For illustration, we have only shown thread `T1` and thread `T2`. For example, the cost of the edge between EOT6 and EOT7 in thread `T2` is 28 clock cycles, which is obtained by adding the costs of all the nodes between these two tick boundaries from the TCCFG of Fig. 3. Once the automata representing the program are obtained, we create a model by composing them synchronously in the model checker. Then, timing analysis can be performed by checking a CTL property, as detailed in Section III-C. In the next section, we extend the above formulation with an efficient path pruning technique that improves both the WCRT tightness and the model-checking scalability.

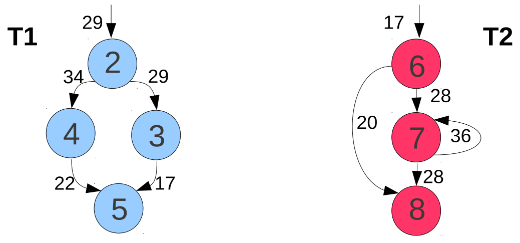


Fig. 4. Abstracted model that captures only tick boundaries.

B. Improving Tightness

In order to find tight WCRT estimates, detection and pruning of infeasible paths is essential [10]. A systematic classification of infeasible paths in synchronous programs has been presented in [10], [18]. They are classified into the following categories: (a) *encoding of tick transition*, (b) *termination and preemption*, (c) *sequentalisation of concurrency in a tick*, and (d) *emit and test of signals*. While the encoding of tick transition involves pruning state-dependent behaviors,

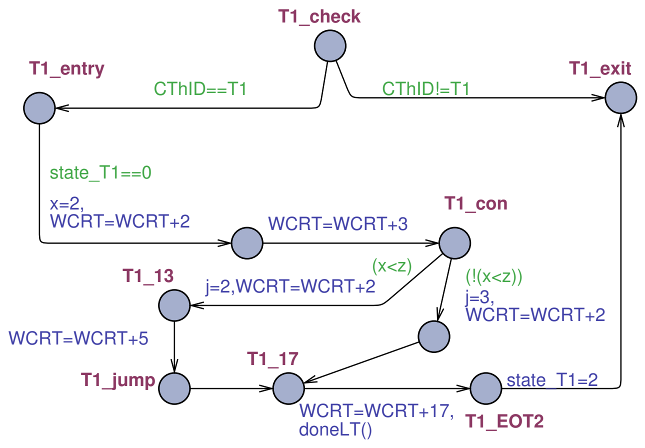


Fig. 5. UPPAAL model of thread *T1*'s first tick of the running example.

the latter three involve some value assignment to a variable followed by a test in the control flow of the program. Hence, we group types (b) to (d) into *set and test* type flow-analysis. In addition to set and test and the encoding of tick transitions, we must also consider *loop bounds*. These three types of infeasible paths are classified as types 1, 2, and 3 as shown in Table I. We further extend these three classifications with types 4, 5, and 6, by introducing more complex data-flow analysis, since the latter types are much more common in C programs. The current ILP based approach can only prune infeasible paths based on simple data-flow analysis (SDFA). They can not analyse data across ticks, and can only handle very simple *set and test scenarios* [18]. This paper presents a much more expressive data-flow analysis (EDFA), analysing more complex *set and test scenarios*, and analysing data across ticks for tighter timing analysis. The rest of the section explains these differences, and the summary is presented in Table I.

Type	Infeasible paths	SDFA	EDFA
(1)	Set and Test	Yes	Yes
(2)	Encoding tick transition	Yes	Yes
(3)	Loops with fixed bounds	Yes	Yes
(4)	Set and Test with expressive data-flow analysis	No	Yes
(5)	Encoding tick transition with expressive data-flow analysis	No	Yes
(6)	Loops bounds requiring expressive data-flow analysis	No	Yes

TABLE I
COMPARISON BETWEEN SDFA AND EDFA BASED PATH PRUNING.

1) SET AND TEST

In Fig. 1, line 7, thread *T1* tests the condition x<z. To evaluate this condition, we need to know the value of variables *x* and *z*. From line 6, we know that the value of *x* will be 2. This can be obtained by applying standard constant propagation to the current *local tick* of the same thread. However, to analyse the value of *z*, we need more global knowledge, i.e., *T1* needs to be aware of the data-flow from the *main* thread. This type of pruning is classified as type-4 in Table I. In the proposed approach, we assign variable *z* with value 3 in the *main* thread. Thanks to these two assignments, the model checker can evaluate the condition x<z on-the-fly.

In contrast, existing ILP approach can only handle very simple *set and test scenarios* (type-1 in our classification) with *assignments* and *conditions* that have constants as their right-hand-side expressions [18], such as:

```
x=7; if(x>3){...}
```

2) ENCODING OF TICK TRANSITIONS

Fig. 4 shows the tick transitions of thread *T1* and thread *T2* respectively. Without taking any data-flow into account, the first column of Table II shows the possible tick transitions obtained by SDFA: e.g., the pair (2,6) means that the second tick of `T1` coincides with the sixth tick of `T2`. SDFA already gives a tighter result than the earlier max-plus based approach [6]. We classify this type of pruning as type-2. However, if we can analyse data across the ticks and also across the threads, this will make possible further pruning of infeasible tick alignments. We classify this type of pruning as type-5. Since model checking approach can handle complex data-flow, we can prune infeasible tick transitions to states (3,7), (4,8), (5,7) and (3,8), as shown in the second column of Table II.

(SDFA)	(EDFA)
(2,6)	(2,6)
(4,7)	(4,7)
(3,7)	(5,8)
(5,7)	
(5,8)	
(4,8)	
(3,8)	

TABLE II
PRUNING INFEASIBLE TICK TRANSITIONS

3) LOOP BOUNDS

In synchronous programs *instantaneous loops* are not allowed, i.e., each loop must execute at least one *EOT* during every iteration. Due to the need for encoding tick transitions, finding an accurate loop bound can prune many infeasible paths. Consider, for example, the *for* loop on line 27 of Fig. 1. In the termination condition *i<j* in *T1*, the value *j* depends on the condition evaluated on line 8 and line 14 of thread *T2*. Also, *j* value decrements with each iteration (line 29). To analyse this loop bound, one must consider data-flow across ticks and threads. We classify this type of pruning as type-6. In our approach, we assign and test variable values on-the-fly. A similar approach to dealing with loops has been presented in [9]. In contrast, current ILP-based approaches assume that the loop bounds are user-guided [18]. This is only reasonable for simple loops that always terminate after a fixed number of iterations. We classify this simple and less expressive loop bound analysis as type-3 in Table I.

C. Complexity

The UPPAAL model of Fig. 5 is automatically obtained from the PRET-C program. This is a simple translation with linear complexity. From this UPPAAL model, we compute the WCRT of the program by model checking a property of the form $AG(gtick \Rightarrow WCRT \leq val)$, where the value of val is determined as follows. We first calculate the $WCRT_{max}$ of the program by summing up the maximum local tick value for every thread. Similarly, the minimum WCRT value, $WCRT_{min}$, is obtained by adding the minimum local tick lengths for each thread. Our estimated WCRT value, $WCRT_{est}$, will

Example	LOC	Observed WCRT	(MAX+) Est WCRT	(SDFA) est. WCRT (1)(2)(3)	(EDFA) est. WCRT (1)to(6)	SDFA/ EDFA
Synchronizer	455	238	608	422	268	1.57
ProducerConsumer	567	259	808	523	294	1.78
Smokers	648	437	1309	903	521	1.73
Channel Protocol	727	644	1426	897	685	1.31
Robot Sonar	1081	764	2028	1688	858	1.97
Synthetic1	1569	898	3593	2127	1022	2.08
Synthetic2	1630	786	3617	1752	942	1.86
Average						1.67

TABLE IV
COMPARING THE WCRT WITH SDFA AND EDFA BASED PATH PRUNING.

lie between the interval $[WCRT_{min}, WCRT_{max}]$. Then, we use standard binary search to find the WCRT. The overall complexity is $O(|M| \times |\phi|)$, where M is the model of the program and ϕ is the complexity of the query. Further details are available in [17].

IV. Results

In this section, we present a set of experimental evaluations to support our qualitative claims. We compare the effectiveness of the proposed technique by comparing the earlier ILP-based method (path pruning with SDFA in Table I) with this paper's method (path pruning with EDFA in Table I). We then present the effect of context sensitive timing analysis [10] on *tightness*, number of *states explored*, and analysis *time*. The benchmarks selected for the following experiments fit entirely on the on-chip memory, with one clock cycle access time. This helps us to clearly compare the infeasible paths in the program without being affected by the memory hierarchy. Also, in the ILP framework of Fig. 2a, the cache analysis is performed in the final stage. In this paper, we compare our approach with ILP based "WCET with program level context" stage. To get a tight estimate, existing hardware aware static timing analyses are highly tailored to a specific processor architecture. This raises the question of how to compare work quantitatively between two research groups? Interestingly, it is not only the hardware but also a difference in compiler that can affect the tightness of the analysis. This is due to the fact that compilers can introduce infeasible paths during compilation [10]. Hence, it is important to use the same processor architecture and apply the same compilation process.

We have selected the MicroBlaze processor [20] along with its compilation tool chain as a common platform. For a given PRET-C benchmark, we generate the assembly files with the *mb-gcc* compiler, and then extract the TCCFGs. Given these TCCFGs, and based on the restrictions of the ILP approach (classified in Table I), we generate two different UPPAAL models: The *SDFA* model with the ability to eliminate simple *conflicting pairs* (1 to 3 of Table I), and the *EDFA* model which can handle more complex data-flow (1 to 6 of Table I). The first column of Table IV lists a set of PRET-C benchmarks followed by the number of lines of C code under analysis. Column five presents the estimated WCRT with *SDFA*, while column six presents the estimated WCRT using *EDFA*. To evaluate the observed WCRT values (third column), we first identify the worst case execution trace using UPPAAL model

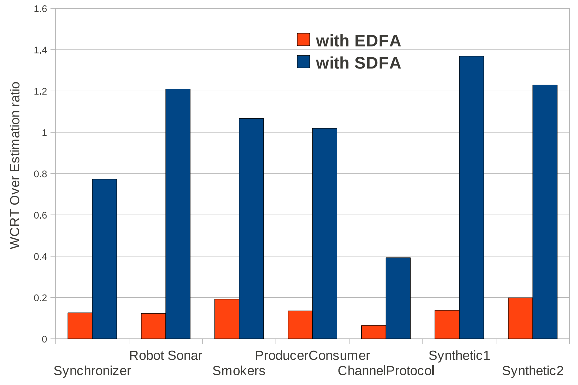


Fig. 6. WCRT over estimation between SDFA and EDFA based pruning.

checker. We then develop the test vectors to elicit this longest path. Then, we run the benchmarks using these paths to get the observed values presented in the third column. The last column shows that, on average, the WCRT estimate is about 67% tighter with EDFA than with SDFA.

We then compare the amount of overestimation. This is done by comparing the observed WCRT (column 3 of Table IV) with the computed WCRT (columns 5 and 6 respectively). The percentage overestimate of the SDFA versus EDFA approach is shown in Fig. 6. On an average, the SDFA approach over-estimates by 89% while EDFA approach only overestimates by about 13%.

To assess the effect of tracking additional context during model checking the following experiment was performed. We randomly classified variables in each benchmarks into four categories. Then, using our tool chain (Fig. 2b), we generate five different UPPAAL models. For the first model, none of the categories in Table I are tracked. The second model tracks all the variables in the first category, the third model tracks the first and the second categories, and the fourth model tracks the first, second and third categories. Finally, the fifth model tracks all the categories. Table III summarizes these experimental results. For each of the five models, we present the *estimated WCRT*, the *time taken* in milliseconds, and the number of *states explored*. Fig. 7a plots the amount of overestimation as the amount of context information increases during WCRT analysis. An increase in context information reduces the number of infeasible paths, thus reducing the over estimation. Interestingly, Fig. 7b shows that the number of states explored decreases as the context information increases. This is due to the fact that the inclusion of more context means that more paths are pruned, leading to fewer number of states explored. This also reduces the time taken to analyse. This can be observed in our largest example, the *robot sonar*. The initial time taken without any context information is about 9.4 seconds; this significantly drops to 0.25 seconds when the entire context information is included.

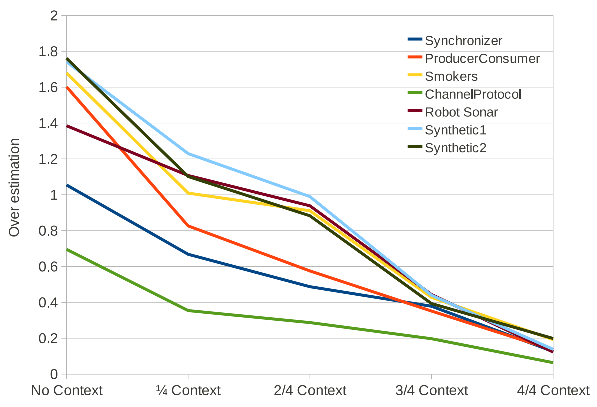
V. Conclusions

Static timing analysis of synchronous programs is critical for validating the synchrony hypothesis for a given environment. In this paper, for the first time, we have proposed an

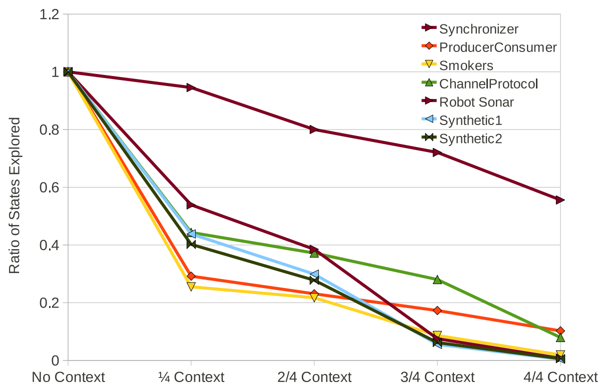
Example	No context			1/4 context			2/4 context			3/4 context			4/4 context		
	Est WCRT	Time taken	States explored	Est WCRT	Time taken	States explored	Est WCRT	Time taken	States explored	Est WCRT	Time taken	States explored	Est WCRT	Time taken	States explored
Synchronizer	487	140	2367	397	156	2238	354	157	1894	328	141	1706	268	141	1316
ProdCons	674	157	4332	473	141	1265	408	156	1000	350	141	749	294	140	444
Smokers	1171	297	41668	878	203	10628	835	203	9051	624	188	3602	521	172	797
ChannelProtocol	1092	969	228479	872	484	101312	829	437	85002	771	360	64064	685	219	18198
Robot Sonar	1822	9407	2489919	1610	5156	1342277	1481	3578	958657	1103	829	187754	858	250	18256
Synthetic1	2462	19423	5149956	2002	8765	2256776	1787	5875	1539491	1293	1234	287228	1022	328	22396
Synthetic2	2170	13203	3501194	1652	5563	1408141	1480	3703	972202	1095	1015	216758	942	297	21418

TABLE III

EXPLORING THE REACTION OF THE MODEL CHECKER WITH MORE CONTEXT INFORMATION



(a) Tightness



(b) Number of States Explored

Fig. 7. Context sensitive information vs tightness and states explored.

approach that can perform WCRT analysis on a concurrent intermediate format. This differs from earlier approaches using integer linear programming (ILP), which compile away the concurrency before deriving the ILP constraints, yielding many complications. We have developed a model checking based formulation that can efficiently deal with concurrent state-spaces. We have also presented a method to prune infeasible paths much more aggressively than existing approaches. In the future, we will extend our current formulation with memory hierarchy.

References

[1] Uppaal tool. www.uppaal.com. last accessed on 12.3.09.

[2] S. Andalam, P. Roop, and A. Girault. Predictable multithreading of embedded applications using PRET-C. *Proceedings of ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July, 2010.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] V. Bertin and et al. Taxys = Esterel + Kronos - a tool for verifying real-time properties of embedded systems. In *IEEE Conference on Decision and Control*, 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*, 203(4):65–79, June 2008.

[7] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, April 1991.

[8] S. A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007. Article ID 52651.

[9] B. Huber and M. Schoeberl. Comparison of implicit path enumeration and model checking based WCET Analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 23–34. OCG, 2009.

[10] L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of Esterel programs. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 870–873, New York, NY, USA, 2009. ACM.

[11] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *CODES+ISSS*, pages 173–178, 2008.

[12] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing analysis of Esterel programs on general-purpose multiprocessors. In *DAC*, pages 48–51, 2010.

[13] F. Laroussinie, N. Markey, and P. Schnoebelen. Model checking timed automata with one or two clocks. In *CONCUR*, pages 387–401, 2004.

[14] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *International Real-Time Systems Symposium (RTSS)*, pages 256–264, Cancun, Mexico, 2003. IEEE Computer Society.

[15] M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, April 2009.

[16] A. Metzner. Why model checking can improve WCET analysis. In *CAV*, volume 3114, pages 334–347, 2004.

[17] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, Grenoble, France, October 2009. IEEE.

[18] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 358–363, New York, NY, USA, 2006. ACM.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[20] Xilinx. *MicroBlaze Processor Reference Guide*, 2008.