Efficient WCRT Analysis of Synchronous Programs using Reachability

Matthew Kuo, Roopak Sinha, Partha Roop
Department of Electrical and Computer Engineering
University of Auckland
mkuo005@aucklanduni.ac.nz, r.sinha@auckland.ac.nz, p.roop@auckland.ac.nz

ABSTRACT

Static computation of the worst-case reaction time (WCRT) is required for the real-time execution of synchronous programs. Existing approaches use model checking or integer linear programming. we formulate this as an abstraction-based reachability analysis yielding a lower worst case complexity. Benchmarking shows a significant overall speed-up of 64-times over existing approaches.

Categories and Subject Descriptors

B.8.2 [**Performance Analysis and Design Aids**]

General Terms

Performance, Reliability, Algorithms

Keywords

Synchronous, Reachability, Worst Case Analysis

1. INTRODUCTION

Real-time systems must meet precise timing requirements in addition to being functionally correct. Synchronous programs are ideal for real-time applications as they divide the program execution into discrete computation blocks, called *ticks* [3]. This leads to simpler temporal semantics than asynchronous systems, and allows efficient static timing analysis. Moreover, all correct synchronous programs guarantee determinism (safety) and reactivity (liveness) making them ideal for real-time system implementation [3].

In order to ensure that all real-time constraints are met, we must find the longest time of any tick, called the *worst-case reaction time* (WCRT) of the system. A synchronous program can guarantee that it meets all timing constraints if its ticks are scheduled using a clock with a period greater than or equal to the WCRT.

Given a synchronous program, the computation of the WCRT involves a trade-off between *the time taken to compute the WCRT* and the *tightness*. In order to reduce the time taken to compute the WCRT, designers use various abstractions that may result in a WCRT value which is an over-approximation of the actual WCRT of the system [7, 10]. On the other hand, computing tighter WCRT values requires including more information about the program, such as data variables, which increases the overall computation time.

DAC 2011, June 5-10, 2011, San Diego, California, USA.

In this paper, we propose a static WCRT computation method based reachability analysis. We show that our technique achieves significant speed-up compared to existing techniques while maintaining the same tightness.

1.1 Related Work

Name	Technique	Notion of Time	Complexity
Max-plus [5]	Linear search	Real-time	Linear in sum of individual thread sizes
NUS [7, 6]	ILP	Real-time	NP Hard
Taxys [4]	Model checking	Logical time	Exponential
UoA MC[10]	Model checking	Real-time	Product of thread sizes× binary search
Proposed	Reachability	Real-time	Product of thread sizes

Table 1: Qualitative comparison of timing analysis techniques for synchronous programs

Tab. 1 summarizes the current WCRT analysis techniques. The earliest work in this field is the max-plus approach [5], where the WCRT of a multi-threaded program is calculated by summing the WCRT of its individual threads. This method has a linear complexity, but WCRT estimates are usually gross over-approximations.

In [7], an integer linear programming (ILP)-based formulation has been presented. First, the synchronous program is compiled into a sequential C program, and then ILP constraints over this sequential program are computed. In [6], this technique is extended to include *tick-alignment* (to prune infeasible paths) by using an additional automaton. The conversion of a concurrent synchronous program into a sequential program, and then using additional information to remove infeasible paths is counter-intuitive. Also, the worst case complexity of ILP is NP-hard.

Some approaches exploit the concurrent nature of synchronous programs to view the WCRT computation problem as a model checking problem. In [10], bounded-integer counting based model checking is presented. A synchronous program is first compiled into machine code to extract its precise timing characteristics. Then, a higher level structure called *Timed Concurrent Control Flow Graph* (TCCFG) is generated, which is then entered into the UPPAAL model checker [2]. The user can then query the model checker with a guess (obtained from a simple heuristic like max-plus) to check if the guess is equal to or less than the WCRT of the system. The guess is then refined by a binary search until the WCRT value is found. This technique is more efficient than the ILP approach [6] because path-pruning does not require additional information, and the timing analysis can be performed on the concurrent program description (contained in the TCCFG). However, the use of binary search means that the model checking has to be carried out in multiple steps.

In Taxys [4], a timed model of an Esterel program is extracted

and is composed with the synchronous model of a non-deterministic environment (modelled by adding a npause instruction) to extract a global model. The technique can be used to check timing properties such as delays between the reception of inputs and the emission of outputs, and throughput constraints such as buffer overflows/underflows. However, this technique has exponential complexity due to the use of real-valued clocks.

In [8], a WCRT computation method for Esterel programs executing on multiprocessor platforms is presented. This technique takes micro-architectural features such as possible cache delays into account. At compile time it pre-computes feasible control states and removes infeasible execution paths. However, it does not allow the tracking of variables during WCRT computation.

In this paper, we show how WCRT computation can be solved by on-the-fly reachability analysis of a synchronous program. The main contributions of this paper are: (1) The proposed method has lesser complexity than model checking; (2) the proposed technique contains thread-level optimizations that are not present in existing techniques, allowing us to significantly speed up the WCRT computation; (3) we allow pruning of infeasible paths using variable tracking between single and multiple ticks for tighter WCRT computation; and (4) we provide extensive benchmarking results that show an overall speed-up of 64-times over model checking [10].

We now present details of our approach. Sec. 2 presents an overview of our approach using a motivating example. Sec. 3 presents the WCRT computation algorithm and associated concepts. Sec. 4 shows how tighter WCRT analysis can be performed using variable tracking. Benchmarking results appear in Sec. 5, and concluding remarks and future directions are presented in Sec. 6.

2. OVERVIEW

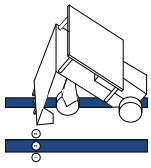


Figure 1: A line-tracking robot

We use the motivating example of a line-tracking robot, as shown in Fig. 1. The robot contains three frontal sensors to detect the location of the line. A sensor returns a value 1 if it is over the line. The thickness of the line allows up to 2 sensors to detect the line at any time. Different combinations of the sensor values adjust the robot back on to the line by turning the robot using the speed differential between its wheels. The speed at which the robot can travel is proportional to the computation time it takes for reading the sensors. This is needed so that the robot can take appropriate corrective measures to continue tracking. In this example, the speed of the robot is dependent on the computation of the system's WCRT.

To describe the above real-time system, we use PRET-C [1], a synchronous language based on C. It is better suited for real-time applications than other synchronous languages since C is the language of choice for most embedded designers. In addition, using PRET-C allows us to directly compare the effectiveness of our approach with the model checking approach presented in [10]. PRET-C extends C by introducing a number of synchronous constructs. For example, it is possible for a thread to spawn new threads, that execute concurrently, using the par construct. In addition, threads have local state boundaries (known as local ticks) marked by their respective EOT nodes. These are synchronization barriers for computation of a global tick of the program.

```
1  int main(void){
2    PAR(Decoder,MotorDriver);
3      if (error 1) set LED 1;
4      if (error 2) set LED 2;
5      else set both LED;
6  }
7  thread Decoder(void) {
8    initialize variables;
9    EOT;
10   while(1) {
11     process sensor data;
12     store previous motor instructions;
13     //set motor control according to sensor data
14     if (sensor==some value) {
15       set motor according to table 2
16     }
17     EOT;
18     if(robot is turning){stuckCounter++}
19     }else{reset stuckCounter;}
20     if(stuckCounter over threshold){
21       set error 2;
22       stop robot;
23     }
24     if(error > 0){break;}
25     else{EOT;}
26   }
27 }
28 thread MotorDriver(void){
29   int PWMCounter = 0;
30   EOT;
31   while(1){
32     if(error > 0){
33       stop robot;
34       break;
35     }else{
36       if(PWMCounter < goLeft){
37         leftMotor = 1; //set left high
38       }else{
39         leftMotor = 0; //set left low
40       }
41       if(PWMCounter < goRight){
42         rightMotor = 1; //set right high
43       }else{
44         rightMotor = 0; //set right low
45       }
46       PWMCounter++;
47       if(PWMCounter == maxSpeed){
48         PWMCounter = 0;
49       }
50       EOT;
51     }
52   }
53 }
```

Figure 2: PRET-C pseudo code of line following robot

Abstracted PRET-C code of the line-tracking robot is shown in Fig 2. The main thread of the program (main) spawns two threads, one each for sensor decoding (Decoder) and motor PWM generation (MotorDriver), using the PAR construct (line 2). The thread main resumes when the other two threads terminate (an error state is reached) (line 3).

Sensor			Motor		Error
Left	Middle	Right	Left	Right	One
0	0	0	0	0	0
1	0	0	0	2	0
0	1	0	1	1	0
1	1	0	0	1	0
0	0	1	2	0	0
1	0	1	0	0	1
0	1	1	1	0	0
1	1	1	0	0	0

Table 2: Look-up table for tracking

Decoder reads sensor values and sets the direction of the robot by setting the motor speed to 0, 1 or 2 for each wheel. This is used to adjust the robot's movement by using a differential drive. Details of how the robot responds based on sensor input is presented in Tab. 2. MotorDriver generates PWM waveforms controlling each wheel by incrementing a PWM counter between 0 and 2. Speed values of 0, 1 and 2 generate 0%, 50%, and 100% duty cycles respectively for the corresponding wheel. The motor thread terminates when an error state is detected by the decoder thread and the output to each wheel is set to 0 (line 29).

2.1 WCRT computation process

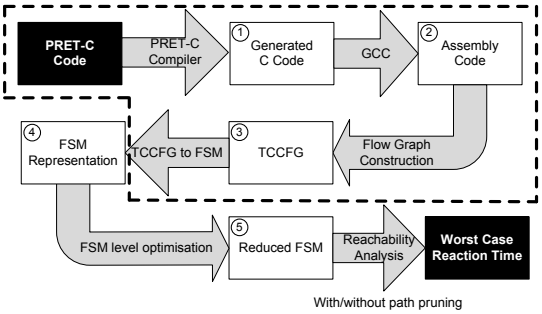


Figure 3: `WCRT` computation process

Fig. 3 shows the process involved in determining the `WCRT` of a PRET-C program. The PRET-C code is compiled using the PRET-C translator (compiler) into C code (step 1), which is then compiled into assembly using `gcc` (step 2). The assembly code, from which accurate timing information about every single instruction can be obtained, is analyzed to create a *Timed Concurrent Control Flow Graph* (TCCFG). This step assumes that speculative features such as memory hierarchy are not used by the program. In other words, the program fits entirely on the on-chip cache of the chosen processor. We choose the MicroBlaze processor [9] to be able to directly compare our approach with the UoA Model Checker (UoA MC) approach [10]. The TCCFG is a monolithic structure that contains information about concurrent control flow and tick boundaries. We follow the process to create a TCCFG as presented in [10].

A readable abstraction of the TCCFG for the line-tracking robot is shown in Fig. 4. The TCCFG has unique *start* and *end* nodes, and *EOT* nodes mark the end of a tick. *Action* nodes correspond to computations, while conditional nodes correspond to branching based on testing the values of program variables. *Fork* nodes describe the concurrent behavior of the program, where each outgoing transition corresponds to a newly spawned thread. When each thread ends, the corresponding join node is entered, allowing the spawning thread to resume execution. Each block has an associated *cost*, which is equal to the number of processor clock cycles needed to execute the corresponding code. The problem of finding the `WCRT` is equivalent to determining the longest execution path between two tick boundaries.

In our approach, the TCCFG is first translated into individual FSMs which correspond to individual threads of the system (step 4). The resulting FSMs contain exactly the same information about each thread as the TCCFG. However, this step allows us to optimize the state space of individual threads before performing reachability. After these optimizations, we compute the synchronous composition of the FSMs, and use a global variable, called `WCRT`, which contains the highest reaction time visited at any stage during reachability (step 5). On termination, this variable contains the highest reaction time of a single tick.

3. WCRT ANALYSIS USING REACHABILITY

3.1 Thread-level processing

The first step involves translating the TCCFG into thread FSMs, each relating to a single thread. This step allows us to perform optimizations that are not done in other techniques [6, 10].

Each `Fork` and `Join` path in a TCCFG relates to a thread. We convert each individual path starting from a `Fork` node into a separate thread FSM. The original `Fork` node is left with a single transition to the corresponding `Join` node. This transition fires when all children thread terminate. The individual FSMs contain the same types of nodes as a TCCFG. The only difference is that `Fork` nodes have only one outgoing transition to a `Join` node.

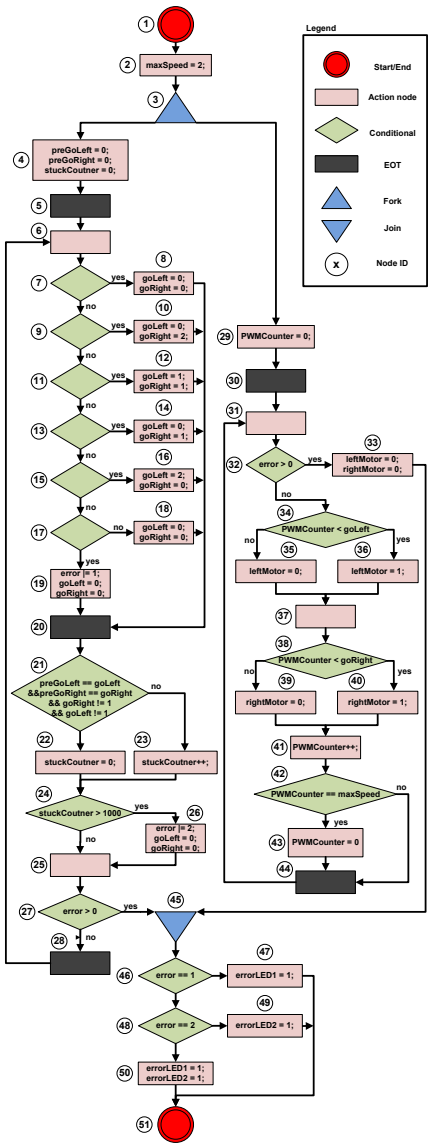
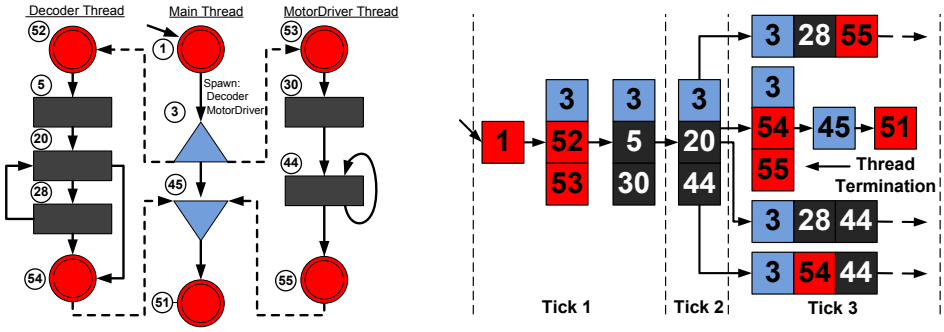


Figure 4: TCCFG of the line-tracking robot

Fig. 5a shows how the TCCFG for the line-tracking robot shown in Fig. 4 is translated into three FSMs. The two concurrent paths, relating to `Decoder` and `MotorDriver` starting from the *`Fork`* node (node 3 in Fig. 4), are converted into individual FSMs. The `main` FSM still contains the *`Fork`* node, but has only 1 transition to the *`Join`* node (node 45). The information about the threads being spawned is added to the transition leading to the `Fork` node.

In addition, we process each FSM by moving the cost associated with nodes in the TCCFG to transitions in the individual FSMs. This step is linear in the size of the FSM being processed, and serves as the basis of the following thread-level optimizations.

3.1.1 Thread-level Optimizations



(a) Translation of TCCFG into FSMs (b) Reachability illustration

Figure 5: Example of optimization and reachability analysis

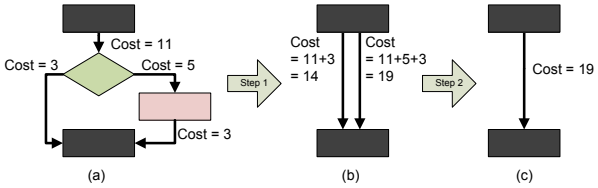


Figure 6: Thread-level optimizations

We perform the following optimizations for each thread:

- **O1: Removal of redundant nodes:** This step involves removing redundant nodes from thread FSMs. For example, each *action* node is removed (shown as step 1 in Fig. 6), and is similar to the TCCFG optimization presented in [10]. Each transition leading to that node is combined with every transition leading out from that node, and the cost of the combined transition is the sum of its constituent transitions. Similarly, each conditional block is removed, and its outgoing transitions are combined with every incoming transition. As the goal of the `WCRT` analysis is to find the largest cost between two `EOT`s, reducing the number of nodes between `EOT`s helps in faster reachability analysis. This optimization removes unnecessary *conditional* and *action* nodes, minimizing the number of nodes in each FSM. This optimization ignores the status of program variables, and hence may result in an over-approximation of the `WCRT`. We address this issue later in Sec. 4.
- **O2: Removal of duplicate transitions:** We prune duplicate transitions by finding all equivalent transitions in a thread, and retaining only one transition that has the largest associated cost (shown as step 2 in Fig. 6). Two transitions are equivalent if they have the same source and destination nodes. We weaken this notion of equivalence later in Sec. 4, so that variable conditions and assignments are also taken into account.

3.2 Reachability using on-the-fly composition

The algorithm uses PRET-C composition to compute the reachable states. In PRET-C, threads that execute in parallel have a fixed order of execution. This order of execution is the order of the spawned threads in the invocation of `PAR`. In line 2 of the PRET-C code in Fig. 2, `PAR(Decoder,MotorDriver)` implies that in every tick, `Decoder` executes before `MotorDriver`. Also, the spawning thread is suspended until all spawned threads execute. We maintain this fixed order of execution while computing the reachable state-space, in order to preserve the semantics of PRET-C.

Consider Fig. 5b which shows a part of the forward reachability computation for the three abstracted threads shown in Fig. 5a. The initial state of the reachability graph relates to the initial node (node 1) of the main thread (`main`). We traverse the reachable state space in a depth-first fashion. Whenever a transition to a *fork* node is seen (for example from node 1 to 3 in `main`), the initial nodes (52 and 53) of the newly spawned threads (`Decoder` and `MotorDriver`) are added to the state tuple. The spawing thread node (node 3 of `main`) is suspended and must wait for the spawned threads to finish. From $(3, 52, 53)$, the two spawned threads make transitions to `EOT` nodes 5 and 30 respectively, marking the end of the first global tick. Note that in any tick, the order of execution is *always* `Decoder` followed by `MotorDriver` (the order in which they are spawned). Next, both threads take transitions to nodes 20 and 44 respectively, marking the end of the second tick.

Now, from $(3, 20, 44)$, there are four possible transitions. Transitions to $(3, 28, 55)$, $(3, 54, 44)$ and $(3, 28, 44)$ indicate the end of the third global tick (as one of the threads reaches an `EOT` node while the other one also reaches `EOT` or finishes execution). We do not further illustrate how reachability continues for these three cases. The fourth successor $(3, 54, 55)$ represents the termination of both spawned threads, which re-enables the `Fork` node 3 to take its lone transition to the `Join` node 45 (shown in Fig. 5a) within the same tick. Then, 45 takes its lone transition to the end node 51, which marks the end of the third global tick.

During the above reachability computation, when a global tick finishes, a global variable `WCRT` (initialized to 0) is compared to the computed tick length. If `WCRT` is smaller than the tick length (sum of the cost of the transitions taken), we update its value. Otherwise, we leave it unchanged. On termination, `WCRT` contains the actual `WCRT` of the program.

Termination of the reachability computation is guaranteed because a previously visited state is never visited again. For example, in Fig. 5b, we can reach state $(3, 28, 44)$ from $(3, 20, 44)$. Now $(3, 28, 44)$ also has a transition back to $(3, 20, 44)$. However, as $(3, 20, 44)$ has already been visited earlier, it is not explored again.

In general, given a set of thread FSMs, we can extend the reachability computation to compute `WCRT` for *any* synchronous language. This extension will involve using the relevant composition operator for the chosen language.

In [10], the UPPAAL model checker uses an extra *global tick thread* to handle termination of spawned threads and resumption of suspended threads. In our case, a global tick thread is not needed, and the details of the parallel composition are handled during the

reachability analysis. During reachability, it is not necessary that every EOT state in a thread is combined with every EOT state in a parallel thread. Only reachable states are analyzed. In [6], an additional automaton is required to ensure this *tick alignment*.

4. INFEASIBLE PATH PRUNING

The reachability algorithm presented in the previous section does not take into account the context of program variables. PRET-C provides thread-safe access to shared variables due to its fixed order of thread execution in every tick. Threads may use variables set by other threads to decide on execution paths, making some paths infeasible, and potentially reducing the WCRT [6]. In this section, we show how our method can be extended to use this information for tighter WCRT analysis by pruning infeasible paths using *variable tracking*.

Each PRET-C program defines a set V of variables, and this information is contained within the TCCFG (as shown in Fig. 4). We allow users to choose to track a subset V_t of V. A user may choose to track no variables ($V_t = \emptyset$), resulting in the same outcome as described in Sec. 3. Alternatively, tighter results can be obtained by tracking one, or all variables. Each tracked variable has a specific type (e.g. int), an initial value, and a finite range. Variable tracking affects thread-level optimizations as well as the reachability computation.

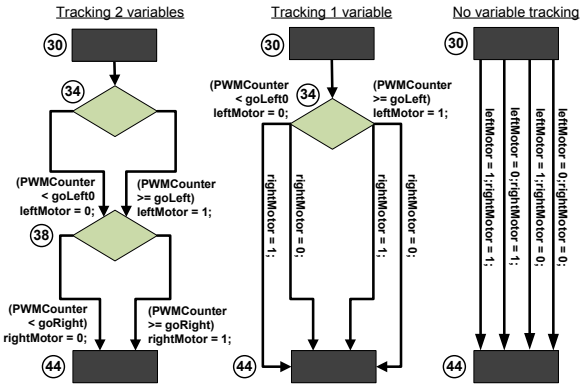


Figure 7: FSMs resulting from tracking different variables

During the pre-processing of individual FSMs (Sec. 3.1.1), we read additional information about test and set operations on tracked variables into the individual thread FSMs. However, all information about non-tracked variables is removed, in order to achieve a compact representation for each thread.

In **O1** (removal of redundant nodes), we do not remove conditional nodes that check the values of tracked variables. Fig. 7 shows a snippet the MotorDriver thread (Fig. 5a). The first conditional node is based on comparing the variables *PWMCounter* and *goLeft*, whereas the second conditional block compares *PWMCounter* with *goRight*. If we track both *goLeft* and *goRight* (along with *PWMCounter*), we retain both conditional nodes from the thread FSM. However, if we track only goLeft, the second conditional node is removed. If both variables are not tracked, we remove both conditional blocks to get only transitions only between EOTs. transitions. We do not allow tracking of variables that depend on any non-tracked variables. For example, given an assignment $v2 = v3$ somewhere in a thread, we do not allow $v2$ to be tracked if $v3$ is not tracked.

For optimization **O2**, we use stricter criteria to prune equivalent

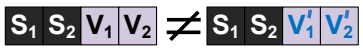


Figure 8: Different notion of state equivalence

transitions. Recall that with no variable tracking, we consider all transitions with the same source and destination nodes as equivalent. However, with variable tracking, two transitions are considered equal when their source and destination nodes are the same, *as well as* when their conditions and assignments are the same too (Fig. 8).

Finally, the reachability algorithm is extended as follows. Each state in the composition is considered to be a tuple of nodes of executing threads *and* a unique valuation for each tracked variable. Two states corresponding to the same thread nodes (in the same order) but variable valuations are considered to be different. Therefore, the EOT to EOT transitions of both states are explored (as shown in Fig. 8).

In [10], the original TCCFG is retained regardless of the number of variables being tracked. This causes the time taken to compute the WCRT to actually *decrease* when more variables are tracked. This is indeed counter-intuitive, because adding information about integer variables with finite ranges should cause the model checker to search over a larger state-space. In our case, we get a gradual increase in the time taken to compute the WCRT, and even when all variables are tracked, the complexity of our approach is still lower than that of [10] due to the absence of binary search.

5. RESULTS

In this section we test our approach using the PRET-C benchmarks presented in [10], along with two larger examples (Synthetic 1 and 2) (Tab. 3). These programs vary from 496 (Synchronizer) to 1630 (Synthetic 2) lines of PRET-C code. These programs fit entirely on the on-chip memory of the MicroBlaze processor [9].

We implement the model checking approach presented in [10] to compare the effectiveness of our approach. We have also extended the approach of [10] with infeasible path pruning to make the two approaches equivalent with respect to capability. Due to the need for binary search, the UoA MC approach takes much longer to execute. For a fairer comparison, we report the time for *only* the final step in the binary search process for each benchmark.

Tab. 3 presents the speed of WCRT analysis for each benchmark with and without variable tracking. Reachability produces the same WCRT values as the UoA MC approach, but achieves significant speed-ups. On average, we observe a speed-up of 320 times when no variable tracking is used. When all variables are tracked, our approach is on average 16.4 times faster than the UoA MC approach.

Fig. 9 shows how the time taken to compute the WCRT varies when we increasingly track more and more variables for the Robot-Sonar example [10]. This program has 1081 lines of code, and contains 7 shared variables. As more variables are tracked, reachability takes steadily longer because the reachable set of states increases by a factor of the ranges of variables being tracked. However, some of this increase is countered by the extra variable information which helps in pruning infeasible paths.

Fig. 10 shows how WCRT computation time varies as more variables are tracked. We make two important observations here. First, the computation time for our approach is always less than the model checking approach for the three largest benchmarks. As more variables are tracked, the performance of reachability is very similar (slightly better) to that of a *single* pass of the model checking approach. This is due to increased infeasible path pruning in the

		No Variable Tracking				Tracking All Variables			
Example	LOC	Calculated WCRT	UoA MC (Single Parse) A) Time Taken(ms)	Proposed B) Time Taken(ms)	Speed-up A / B	Calculated WCRT	UoA MC (Single Parse) A) Time Taken(ms)	Proposed B) Time Taken(ms)	Speed-up A / B
Synchronizer	455	487	140	1	140.00	268	141	7	20.14
ProdCons	567	674	157	1	157.00	294	140	3	46.67
Smokers	648	1171	297	1	297.00	512	172	4	43.00
ChannelProtocol	727	1092	969	9	107.67	685	219	33	6.64
Robot Sonar	1081	1822	9407	13	723.62	858	250	170	1.47
Synthetic 1	1569	2462	19423	32	606.97	1022	328	236	1.39
Synthetic 2	1630	2170	13203	64	206.30	942	297	127	2.34

Table 3: Comparison of Time taken for WCRT computation

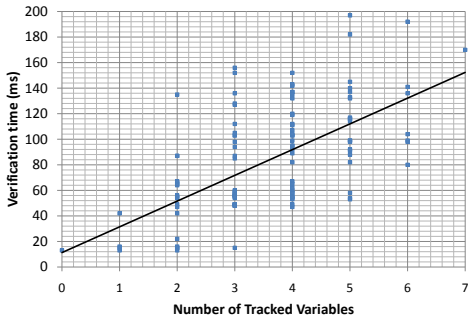


Figure 9: `WCRT` computation time vs no. of variables tracked

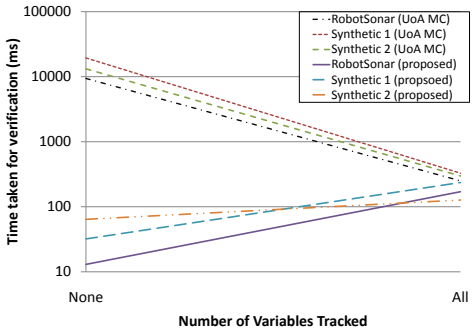


Figure 10: Computation time trends

model checking method, and a larger state space to be processed by our method.

These results also show the power of the thread-level optimizations which are not done in the UoA MC approach. With no variable tracking, our thread-level optimizations significantly reduce the state-space for faster reachability. The number of nodes and transitions removed during thread-level optimizations decreases as more variables are tracked, resulting in a gradual increase in computation times. For the UoA MC approach however, as more variables are added, the computation time decreases due to increased infeasible path pruning.

6. CONCLUSIONS

In this paper, we presented a static analysis method based on reachability, and aided with thread-level optimizations, to compute the worst-case reaction time, or `WCRT`, of synchronous programs. Even though reachability has lower complexity, we achieve the same tightness in `WCRT` computation as more complex techniques such as model checking. For the same benchmarks, our technique significantly out-performs model checking by an overall speed-up of 64-times. This approach is also more general, as it can be extended to other synchronous languages by simply using different synchronous parallel operators during reachability.

Future directions include the use of abstract interpretation for more efficient variable tracking, and the creation of models for speculative features such as caches and pipelines to allow more accurate `WCRT` analysis.

7. REFERENCES

[1] S. Andalam, P. S. 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.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, Sept. 2004.

[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] 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.

[5] 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.

[6] L. Ju, B. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of esterel programs. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 870 –873, July 2009.

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

[8] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing analysis of esterel programs on general-purpose multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 48–51, New York, NY, USA, 2010. ACM.

[9] Micrium. *uC-OSII and Xilinx Microblaze processor, Application Note AN-1013*, 2008.

[10] 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.