ILP$_c$: A novel approach for scalable timing analysis of synchronous programs

Jia Jie Wang
University of Auckland
jwan232@aucklanduni.ac.nz

Partha S. Roop
University of Auckland
p.roop@auckland.ac.nz

Sidharta Andalam
TUM CREATE
sidharta.andalam@tum-create.edu.sg

ABSTRACT
Synchronous programs have been widely used in the design of safety critical systems such as the flight control of Airbus A-380. To validate the implementations of synchronous programs, it is necessary to map the program’s logical time (measured in logical ticks) to physical time (the execution time on a given processor). The static computation of the worst-case execution time of logical ticks is called Worst Case Reaction Time (WCRT) analysis. Several approaches for WCRT analysis exist: max-plus algebra, model checking, reachability and integer linear programming (ILP). Of these approaches, reachability, model checking and ILP provide reasonably precise worst case estimates at the expense of longer analysis time. Apart from max-plus based approaches, which can produce large overestimates, the existing approaches suffer from the state space explosion problem. In this paper, we develop a new ILP based approach, called ILP$_c$, which exploits the concurrency explicitly in the ILP formulation to avoid the state space explosion problem. Through extensive benchmarking we demonstrate the efficacy of the approach: for complex programs, ILP$_c$ often produces orders of magnitude faster compared to the existing approaches, while achieving same level of precision. Thus, this paper paves the way for scalable WCRT analysis of complex embedded systems designed using the synchronous approach.

Categories and Subject Descriptors
C.3 [Computer Systems Organization]: Special purpose and application based systems—Real-time and embedded systems

General Terms
Languages, Verification, Performance

Keywords
Integer linear programming, Scalability, Static timing analysis, Synchronous languages

1. INTRODUCTION
Synchronous languages are ideal for designing safety critical applications as they are based on a formal model of computation [5]. This leads to a simpler temporal semantics, which guarantees deterministic concurrency and consequently facilitates the verification of safety properties. The synchronous semantics divides the execution of a program into a sequence of discrete instants, called ticks, during which all events are synchronized. Since the environment is sampled only once in each tick, it is essential to ensure that the worst case execution time of a tick is less than the minimum inter-arrival time of events from the environment (timing correctness). This is known as the synchrony hypothesis [5], and the static computation of the worst case execution time of a tick is known as Worst Case Reaction Time (WCRT) analysis [8, 7, 15].

In [12], an Integer Linear Programming (ILP) based WCRT analysis is proposed. This approach first compiles a synchronous program into a sequential control flow graph (SCFG), from which the sequential C code is generated. Then the conventional ILP formulation for SCFG is applied to compute the WCRT. Subsequently, they extended this work with an additional tick transition automaton in [11], to improve the precision by pruning infeasible paths. A model checking based approach for WCRT analysis is developed in [4]. The program is modelled as a set of interacting finite state automata that are synchronously composed. Computing the WCRT is formulated as a model checking question. This approach has the ability for more complex infeasible path pruning than the ILP approach, and can often provide tighter WCRT estimates as illustrated in [4]. Recently, a reachability based WCRT analysis is proposed in [13] which achieves the same precision as the model checking approach in [4] while having shorter analysis time.

All of the presented approaches are adaptations of existing worst case execution time analysis techniques, that were developed primarily for sequential programs [16], and may not scale for programs with a large number of concurrent threads (see Sec. 2). More recent work in [14] focuses on the timing analysis of concurrent programs on multi-core architectures. The behaviour of each thread on each core is represented using a matrix. All possible inter-leavings and synchronisation points between two threads (two matrices) can be computed using the Kronecker product. The authors assume that all threads are always active and there is no hierarchy of threads. In addition, only small examples (control flow graph with less than 10 nodes) are presented. Thus, it is hard to validate the scalability of the approach for non-trivial programs.

In this paper, we propose a new ILP based approach for WCRT analysis, called ILP$_c$, which is suitable for large multi-threaded synchronous programs. The key idea is to consider the concurrency and synchronization inherent in the synchronous programs, so as to derive appropriate ILP constraints. As a consequence, we not only avoid the state-space explosion problem but also produce results that are as precise as the existing approaches [4, 11, 13]. Benchmarking demonstrates that, for large programs, the proposed approach is or-
2. MOTIVATING EXAMPLE

![Figure 1: FSM representation of a 3-threaded synchronous program.](image1)

In a synchronous program, concurrent threads execute in a lock-step manner. During a tick, each participating thread executes until it reaches an \texttt{EOT} (end of tick) or a \texttt{pause} statement, and threads will only advance to the next tick when all the participating threads have reached their respective \texttt{EOT}/\texttt{pause} statements. The \texttt{EOT}/\texttt{pause} statements could be considered as state-boundaries of a thread. Consequently, threads can be represented in an abstracted manner as Finite State Machines (FSMs). During each tick of the program, all FSMs will take exactly one transition. Fig. 1 shows an example of three concurrent threads, with execution costs, in processor clock cycles, labelled beside each state. An example execution trace is shown in Fig 2. During the first tick (\textit{tick 1}), the FSMs T1, T2 and T3 execute states A1, B1 and C1 respectively. The execution continues in a tick-by-tick fashion. We use this motivating example (Fig. 1) to illustrate the distinction between the different approaches. The existing approaches for WCRT analysis can be broadly classified into two categories, either \textit{summation} based or \textit{state exploration} based.

2.1 Summation Approaches

Max-plus [8] and conventional ILP are summation based approaches, which compute the WCRT by summing up the worst case execution time of each thread. Applying a summation based approach to the motivating example results in a computed WCRT of 40 clock cycles.

\[
\text{WCRT} = \text{MAX}(T1) + \text{MAX}(T2) + \text{MAX}(T3) = A3 + B1 + C2 = 20 + 10 + 10 = 40
\]

However, the \textit{combination of states} A3, B1 and C2 is \textit{infeasible} as B1 and C2 will never execute together in the same tick. Hence the computed WCRT will never occur during actual execution. This mismatch of states between concurrent threads is known as the \textit{tick alignment} problem [15].

2.2 State Exploration Approaches

On the other hand, the techniques such as [4, 11, 13] resolve the tick alignment problem by modelling the tick transitions of the program, thereby computing the execution time of all the feasible combinations of states. Applying a state exploration based approach to the motivating example results in a computed WCRT of 35 clock cycles.

\[
A1 + B1 + C1 = 25; \quad A2 + B2 + C2 = 28; \\
A3 + B1 + C1 = 35; \quad A1 + B2 + C2 = 23; \\
A2 + B1 + C1 = 30; \quad A3 + B2 + C2 = 33;
\]

\[
\text{WCRT} = \text{MAX}(25, 28, 35, 23, 30, 33) = 35
\]

State exploration based approaches can achieve better precision compared to summation based approaches at the expense of longer analysis time. This trade-off between precision and scalability presents a challenge in computing precise WCRT for large multi-threaded synchronous programs. However, consider the case of revising the program where the cost of B2 increases to 15. In this case, summation based approaches would produce the same WCRT as state exploration based approaches, while having a much faster analysis time, since the computed worst case coincides with aligned ticks (A3, B2 and C2). This shows that summation based approaches are not always less precise. Based on this observation, we develop an alternative approach using ILP that is tailored for large synchronous programs. It uses an idea similar to iterative Counterexample Guided Abstraction Refinement [9] to avoid the exploration of the complete state-space, thereby a better amortised analysis time compared to existing approaches, while achieving the same level of precision as the existing state exploration based approaches.

2.3 Our Approach (ILPc)

We first develop an ILP model that computes the WCRT without considering tick alignment. We consider this longest execution path and then check whether the ticks in this path actually align. If the verification returns a positive answer, we have computed the final WCRT. If, on the other hand, the verification returns a negative answer (e.g., A3, B1 and C2 in Fig. 1), we refine the ILP constraints to reflect this infeasible combination of states, then recompute a new WCRT and repeat the verification of tick alignment. The analysis terminates when the verification returns a positive answer. We envisage that, for real-life benchmarks, ILPc can return a positive verification result without having to explore the full state-space. The central tenet of the paper is founded on the above hypothesis of amortizing the performance of ILP for concurrent programs through this iterative refinement process.

The main contributions of this paper are as follows.

- We propose an iterative WCRT analysis framework, which produces tight estimates and is scalable for large synchronous programs. Unlike the existing approaches, where analysis must be completed in order to produce a safe WCRT estimate, the proposed framework can be stopped at any time after the initial iteration of analysis and still able to produce a safe WCRT estimate.

- We extend the conventional ILP technique for SCFG to model TCCFG [3]. Our approach directly models the semantics of synchronous programs: logical ticks, concurrent threads, and preemptions.

- Orthogonal to the control flow graph based ILP model, we developed another ILP based technique for verifying the tick alignment of a given execution path. This technique along with the ILP model achieves precise WCRT estimations.
3. PRET-C AND TCCFG

In this section, we briefly discuss about (1) the execution semantics of a synchronous programming language, (2) a method for capturing the timing properties of an underlying architecture and (3) an intermediate format that captures the execution semantics of the program and the timing properties of the execution platform in a single control flow graph.

Synchronous languages: Esterel [6] and SCADE [2] have been successfully used for the design of safety-critical embedded systems. However, C remains the language of choice for most embedded applications. Hence, we select a recent synchronous C variant called PRET-C [3]. PRET-C is based on a set of simple macros that extends C with synchronous concurrency and preemption. In this paper, we formulate ILP approaches using the intermediate format of PRET-C called TCCFG [3]. Our approach, however, is generic and applicable to other synchronous languages.

Modelling the underlying architecture: To compute the execution time of a program, one must model the behaviour of hardware components such as caches with their replacement policies, and pipelines with their control and data hazards [10]. In most static analysers, the hardware components are first analyzed to compute the worst case execution time of each instruction. This process is known as micro-analysis. After micro-analysis, a Control Flow Graph (CFG) of the program is generated where the nodes are annotated with execution costs. This annotated CFG becomes the input for macro-analysis, where the WCRT of the program is computed. In this paper, our emphasis is on macro-analysis. Thus, for the rest of the section we briefly describe our execution time annotated control flow graph which becomes the input to the macro-analysis presented later in Sec. 4.

Intermediate format: TCCFG consists of fork and join nodes for capturing the concurrency. Abort start and abort end nodes for capturing preemption. EOT nodes for capturing the tick boundaries of a thread. In addition, there are conventional nodes such as computation and condition.

Table 1: Tick snapshot of the running example

<table>
<thead>
<tr>
<th>Tick</th>
<th>Execution path</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E0 → E1 → E2 → E3 → E6 → E13</td>
<td>T1 and T2 terminate</td>
</tr>
<tr>
<td>2</td>
<td>E3 → E2 → E9 → E10 → E12 → E16 → E17 → E19</td>
<td>T1 terminates</td>
</tr>
<tr>
<td>3a</td>
<td>E3 → E2 → E9 → E10 → E12 → E16 → E17 → E19</td>
<td>T1 terminates</td>
</tr>
<tr>
<td>3b</td>
<td>E3 → E2 → E9 → E10 → E12 → E16 → E17 → E19</td>
<td>T1 terminates</td>
</tr>
<tr>
<td>3c</td>
<td>E1 → E4 → E21</td>
<td>Abort condition evaluates to true</td>
</tr>
</tbody>
</table>

Fig. 3 presents an example TCCFG, which will be used to demonstrate ILP. Each node is labelled with a unique ID prefixed with ‘E’, and each edge is labelled with a unique ID prefixed with ‘T’. The number beside each node represents the execution cost of the node on a given execution platform. In this paper, these costs are derived by using a well known approach [10].

4. THE ILP FORMULATION

Fig. 4 shows the overview of ILP. Two models are first generated from the TCCFG by processes 1 and 2, that are called ILPbase and tick expressions. ILPbase is an ILP model that captures the control flow of the TCCFG, while the tick expressions are mathematical expressions that capture the tick transition information. After generating these two models, the analysis enters a process of iterative refinement (processes 3-5). Similar to the convention ILP technique, during process 3, ILPbase is solved to obtain a WCRT estimate, and its
corresponding execution path is extracted from the variable values in the solution. Subsequently, this execution path is verified against the tick expressions in process 4. If the participating ticks in this execution path can be aligned, the analysis finishes, and the computed WCRT is the final WCRT. Otherwise a new ILP constraint is generated in process 5 and is appended to ILP\text{base} to eliminate the infeasible tick combination. Then the analysis starts over again from process 3. The following subsections will describe these processes in details as we analyse the WCRT of the example TCCFG in Fig. 3.

4.1 Process 1: Formulation of ILP\text{base}

While the conventional ILP formulation for synchronous programs [12] works over a SCFG, we formulate ILP\text{base} to work over a TCCFG. This enables us to directly capture ticks, concurrency and preemptions of a synchronous program through modelling the execution of EOT nodes, fork and join nodes, and abort start and abort end nodes, and thereby improve the precision of the ILP model. The tick transition information is not captured in ILP\text{base}, as it requires exploring the full state-space in order to generate all the ILP constraints [11].

The formulation for ILP\text{base} is based on the same modelling principle as the conventional ILP technique for SCFG. Generating ILP\text{base} is achieved in two steps, which are (1) describing the calculation of the total execution cost in the objective function, and (2) modelling the control flow using ILP constraints. Same as the conventional ILP technique, the objective function of ILP\text{base} is formulated as follows:

$$\text{Objective function} = \sum_{i=1}^{n} E_i \times c_i$$

In the formulation, \(n\) is the total number of edges in the TCCFG, \(E_i\) is the variable that represents an edge and \(c_i\) is the execution cost that is associated with the edge. In general, the execution cost of each node is associated with all its inflow edges(s), because a node executes when the control flow reaches it. However, join nodes are exceptions. A join node can receive multiple active inflow edges from concurrent threads during a tick, and it can execute just once, or may not execute at all depending on whether all child threads have terminated. Hence, the execution cost of each join node is added to its outflow edge. For example, the cost of E15 is added to E20 instead of E12 and E19 (i.e., both have 0 cost), and the associated cost of E20 is 15 (i.e., 0+15).

We extend the formulation of ILP constraints of the conventional ILP technique to model the complex control flow in TCCFGs, such as preemption. The conventional control flow at computation and condition nodes are captured using the conventional ILP technique. To simplify the required analysis and enable better utilization of inequality in ILP constraints, the following characteristics of synchronous programs and ILP models are taken into account in the formulation.

- Synchronous programs forbid instantaneous loops and recursion [5], so each edge can only be active at most once during a tick. Hence, the values of the integer variables in the ILP constraints, which represent the edges in the TCCFG, are binary (\(E_i \in \{0,1\}\)). Each edge can be either inactive (‘0’) or active (‘1’).
- Since the objective function is being maximized (i.e., computing for the worst case time) and all the edges have a positive contribution to the objective function (i.e., the execution costs of all nodes are non-negative integers), the ILP solver will set the values of the variables to ‘1’, whenever possible.

The following sections will illustrate the extended formulations for ILP constraints, which are specifically developed for synchronous programs. In our formulation, we define the following two terminologies:

- The outflow edge of an EOT node or the start node will be called an EOT edge. For example, E0, E3 and E7 are EOT edges.
- For a thread named \(T_n\), let EOT\(_T_n\) represents all the EOT edges of this thread. For example, in the example TCCFG (Fig. 3), EOT\(_{T0}\) consists of E0 and E22, and EOT\(_{T1}\) consists of E7 and E9.

4.1.1 Tick constraints

An execution path of one tick always start from EOT edges, and end at EOT nodes, hence we can emulate the execution of one tick by establishing constraints between EOT edges. The ILP constraints should enforce the following 3 behaviours.

1. Each thread can have at most one active EOT edge.
2. All the EOT edges in the child threads should be inactive if their parent thread has an active EOT edge.
3. If a thread has more than one fork/abort start node in sequence or under different branches, only one of them can have active EOT edges in its child threads.

To capture the above behaviours, the ILP constraints are formulated from the global view of the TCCFG. The EOT edges, out of which at most one can be active during a tick, are classified into one group. And for each group, the following ILP constraint is generated to ensure at most one of the EOT edges in the group can be active.

$$\sum \text{EOT edges in a group} \leq 1$$

EOT edges can be effectively grouped by analysis the tick hierarchy of the TCCFG.
In this example, we can only illustrate how to generate ILP constraints to enforce behaviours 1 and 2, as there is no thread that has more than one fork/abort start node (requirement of behaviour 3). If the example TCCFG has a thread, which has more than one fork/abort start node, additional ILP constraints have to be generated to enforce behaviour 3, using similar grouping technique. The ILP constraints should mutually exclude the execution of child threads which are under different fork/abort start nodes from the same thread. While not introducing restrictions for child threads that can execute concurrently.

4.1.2 Fork constraints

Concurrent threads are enclosed by a pair of fork and join nodes. Two sets of ILP constants are formulated to model the forking and joining of concurrent threads, one for modelling the control flow at the fork node (Sec. 4.1.2) and the other one for modelling the control flow at join node (Sec. 4.1.3).

When the control flow reaches a fork node, all its outflow edges should be active to represent the spawning of child threads. This is modelled using the following ILP constraints, which are generated with respect to each fork node.

\[ \forall n \in \text{fork nodes}, \forall E_{in} \in \text{outflow edges of } n \]
\[ E_{out} = \sum E_{in} \]
\[ E_{in} \in \text{inflow edges of } n \]

For example, for the fork node B4 in Fig. 3, the following ILP constraints are generated.

\[ \text{B4: } E_6 = E_5,E_{13} = E_5 \]

4.1.3 Join constraints

A join node executes when the last child thread between a fork-join pair (e.g., B4 and B15) terminates. For example, the join node B15 executes (i.e., E20 is active) when both T1 and T2 have terminated. It is not essential for all child threads to terminate in the same instant (tick) and they may terminate during different instants. However, the instant of execution of a join node always coincides with the instant when the last child thread of the associated fork-join pair terminates. To capture this behaviour, the formulation focuses on constraining the outflow edge of a join node. Each join node has only one outflow edge, and the outflow edge is inactive if any of the following conditions is satisfied.

1. If all inflow edges of a join node (e.g., E12 and E19 for B15) are inactive, the outflow edge should be inactive.
2. If any child thread (e.g., T1 for the join node B15) has an active edge, and its execution does not reach the join node (e.g., E12 is inactive for T1).

In all other cases, the outflow edges of join nodes should be active. To model the first condition, the following ILP constraint is generated with respect to each join node.

\[ \forall n \in \text{join nodes} \]
\[ E_{out} \leq \sum E_{in} \]
\[ E_{in} \in \text{inflow edges of } n \]

Based on the settings that are described at the beginning of Sec. 4, \( E_{out} \) will be set to ‘0’ if none of the edges in \( E_{in} \) are active, and will be set to ‘1’ otherwise. For example, the following ILP constraint is generated to model the first condition for the join node B15.

\[ \text{B15: } E_{20} \leq E_{12} + E_{19} \]

To capture the second condition, ILP constraints are generated from the global view of the TCCFG. For each nested child thread inside the fork-join pair (e.g., T1 and T2 for the B4 and B15 pair), the following ILP constraint is generated.

\[ (1 - \sum E_{in} - E_{spawn}) + \sum E_{terminate} \geq E_{out} \]
\[ \text{where } E_{spawn} \in \text{The outflow edge of } f \text{ that spawns } t \]
\[ E_{terminate} \in \text{The inflow edges of } j \text{ that are from } t \]
\[ E_{out} \in \text{The outflow edge of } j \]

The left hand side of the ILP constraint consists of two parts, which are enclosed by two pairs of brackets. The first part of the ILP constraint captures whether a child thread has active edges. In the scope of a thread, all its execution paths can only start from the edge that spawns it (e.g., E6 for T1), or the EUT edges inside them (e.g., E7 and E9 for T1). Hence we can use these edges to represent all the edges in a thread (i.e., if none of these edges are active, no edges in the thread is active). Also, only one of these edges can be active at most (Sec. 4.1.1). The value of the first part is ‘1’ if and only if all the edges in the thread are inactive; ‘0’ otherwise.

The second part of the ILP constraint captures thread termination. The value of the second part is ‘1’ only when the execution in the child thread reaches the join node (e.g., E12 is active in T1), ‘0’ otherwise. The outflow edge of the join node is forced to be inactive if both parts of the ILP constraint are ‘0’, otherwise it will be set to ‘1’ by the ILP solver. As an example, the following ILP constraints are generated for the example TCCFG to capture the second condition.

\[ T1: (1 - (E7 + E9) - E6) + (E12) \geq E_{20} \]
\[ T2: (1 - (E14 + E16) - E13) + (E19) \geq E_{20} \]

4.1.4 Abort start constraints

In a TCCFG, preemption is captured by a pair of abort start and abort end node. To capture the behaviour of pre-emption, three sets of ILP constraints are generated, (1) for modelling the control flow at an abort start node (Sec. 4.1.4), (2) for modelling the control flow at an abort end nodes (Sec. 4.1.5) and (3) for modelling the preemption of threads (Sec. 4.1.6).

An abort start node is similar to a fork node, which always spawns two threads, a checkabort thread and a abort-Body thread. The priority of these two threads depends on the type of abort: strong or weak. For the TCCFG in Fig. 3, the checkAbort thread has the higher priority than the abort-Body thread as it is a strong abort. In the following formulation, these two threads will be referred as the higher priority thread and the lower priority thread respectively. To capture the control flow of an abort start node, the following ILP constraints are generated.

\[ \forall n \in \text{abort start nodes} \]
\[ E_{high} = \sum E_{in} \]
\[ E_{low} \leq \sum E_{in} \]
\[ \text{where } E_{in} \in \text{inflow edges of } n \]
\[ E_{high} \in \text{the outflow edge of } n \text{ that spawns the higher priority thread} \]
\[ E_{low} \in \text{the outflow edge of } n \text{ that spawns the lower priority thread} \]

When the control flow reaches an abort start node, the higher priority thread is always spawned and executed. Hence the equal sign is used in the ILP constraint. The lower priority thread is only spawned and can execute if it is not preempted by the higher priority thread. Hence the less or equal sign is used, which allows the ILP constraints in Sec. 4.1.6 to set the value of \( E_{low} \) to ‘0’ when preemption takes place. Otherwise \( E_{low} \) will be set to ‘1’ by the ILP solver.

For example, the following ILP constraints are generated
4.1.5 Abort end constraints

The control flow at an abort end node is similar to a computation node. The outflow edge of an abort end node is active if any of its inflow edges is active. This control flow can be captured by the following ILP constraint.

\[ \forall n \in \text{abort end nodes} \]
\[ E_{out} = \sum E_{in} \]
where \( E_{out} \) is the outflow edge of \( n \) and \( E_{in} \) is the inflow edges of \( n \)

This ILP constraint implies that at most one of the inflow edges of an abort end node can be active, as the values of all variables (e.g., \( E_{out} \) in the ILP constraint) are bounded to be either ‘0’ or ‘1’. This is a correct implication. Although the inflow edges of an abort end node are from two concurrent threads (e.g., \( E4 \) and \( E20 \) for \( B16 \) from checkAbort and abortBody), they cannot be active together as preemption takes place when any of them is active. As an example, the following ILP constraint should be generated for the \( B16 \) node in Fig. 3.

\[ B16: \quad E21 = E4 + E20 \]

4.1.6 Preemption constraint

Preemption takes place when the control flow reaches an abort end node. According to the execution semantics of TCCFG, only the higher priority thread can preempt the lower priority thread. The higher priority thread always executes first and only switches to next the thread when it finishes its execution in a tick.

For example, when the higher priority thread in the abort (e.g., checkAbort in Fig. 3) reaches an abort end node (e.g., \( E4 \) is active), all the edges in the lower priority thread (e.g., abortBody) should be inactive, including its nested child threads, as they are preempted. To capture this preemption behaviour, the following ILP constraint is generated for the lower priority thread and each of its nested child threads.

\[ \forall t \in \text{the lower priority thread and its nested child thread between the abort start node}\ s \text{ and the abort end node}\ e. \]
\[ \sum (E_{out}^t) + E_{spawn} \leq 1 - E_{exit} \]
where \( E_{spawn} \) is the outflow edge of \( s \) that spawn the lower priority thread
\( E_{exit} \) is the inflow edge of \( e \) that is from the higher priority thread

Similar to the formulation for join nodes in Sec. 4.1.3, all the edges in the lower priority thread are captured by using the edge that spawns the lower priority thread (e.g., \( E5 \) which spawns abortBody) and EOT edges, as in the scope of the lower priority thread, all the execution paths start with these edges. As an example of the formulation, the preemption in the example TCCFG is modelled by the following ILP constraints.

\[ T1: \quad (E7 + E9) + E5 \leq 1 - E4 \]
\[ T2: \quad (E14 + E16) + E5 \leq 1 - E4 \]

4.2 Process 2: Formulation of tick expressions

Formulating tick expressions is a key contribution of this paper. They are derived orthogonally to perform the verification of tick alignment over the longest tick computed by ILPbase. We use the idea from affine functions in mathemat-
After solving $\text{ILP}_\text{base}$ (process 3 in Fig. 4), the computed worst case execution path is checked against the tick expressions for tick alignment. Fig. 7 shows the overview of the verification process. We first identify the EOT and join nodes in the execution path by checking the their outflow edges. This is then used to extract their tick expressions. An ILP model called $\text{ILP}_\text{check}$ is subsequently generated based on the extracted tick expressions. $\text{ILP}_\text{check}$ is used to find a reference tick count that is to be included in all the extracted tick expressions. This will help to determine whether the combination of EOT nodes in the execution path is feasible. For example, solving the $\text{ILP}_\text{base}$ of the example TCCFG will produce an execution path that has the EOT nodes B3, B5 and B12. The following $\text{ILP}_\text{check}$ is then used to verify whether these EOT nodes can actually align during execution.

\[
\begin{align*}
B3: & \quad xN_1 + 1 \times N_3 = RTC \\
B5: & \quad xN_3 + 2 \times N_4 = RTC \\
B12: & \quad 2 \times N_5 + 2 \times N_6 = RTC \\
& \quad N_1 = 1, N_3 = 1, N_5 = 1 \\
& \quad RTC > = 2
\end{align*}
\]

Here, $N_1$ to $N_6$ and RTC (reference tick count) are non-negative integer variables used in $\text{ILP}_\text{check}$. The largest min of all the tick expressions is ‘2’. Hence RTC must be equal greater than ‘2’. The purpose of $\text{ILP}_\text{check}$ is to determine if there is a solution for the given ILP constraints. The cost of each variable is assigned to ‘1’, as they have no actual effect and meaning in $\text{ILP}_\text{check}$ and an optimal solution is not required. The ILP solver should be stopped as soon as a solution is found. The above $\text{ILP}_\text{check}$ has no solution, as the set of numbers for B5 (1, 3, 5...) does not have any number in common with the set of numbers for B12 (2, 4, 6...). Hence, we can conclude that the EOT nodes B3, B5 and B12 are infeasible during execution.

### 4.4 Process 5: New ILP Constraint

When an infeasible execution path is detected (e.g., the verification failure example in the previous section (Sec. 4.3)), a new ILP constraint will be generated and be appended to $\text{ILP}_\text{base}$ to eliminate the detected infeasible tick alignment. The new ILP constraint is generated as follows:

\[
\sum \text{EOT} \leq D - 1
\]

where $EOT \in$ The EOT edges in the execution path $D \in$ The total number of EOT edges in EOT

For the example in the previous section, the new ILP constraint is:

\[
E3 + E7 + E16 \leq 2
\]

### 5. RESULTS

In this section, we present a comparison of $\text{ILP}_C$ relative to the existing ILP based [11], model checking based [4] and reachability based approaches [13]. The ILP approach in [11] which is based on the Esterel language is reimplemented, and we refer this as $\text{ILP}_s$. $\text{ILP}_s$ takes a SCFG as input, which is obtained from the TCCFG of each benchmarks. In this benchmarking, the open source ILP solver Ip_solve [1] is used to solve all the ILP models for $\text{ILP}_c$ and $\text{ILP}_s$. The break-at-first feature is enabled when solving $\text{ILP}_\text{check}$ models in $\text{ILP}_c$. The model checking-based and the reachability based approaches are originally based on TCCFGs, and the exact same tool chain as in [4] and [13] are directly used in this benchmarking.

### 5.1 The Benchmark Process

The aim of the benchmarking is to evaluate the scalability (analysis time and precision) of the proposed approach relative to the existing approaches. The benchmarking was evaluated on a Windows based machine using an Intel i7 820QM processor with 8 GB of RAM. The evaluation consists of two phases. In the first phase, we evaluate the performance of these four approaches using two sets of synthetic benchmarks. In the second phase, we compute the WCRTs for a set of real-life synchronous programs using the four approaches.

#### 5.2 First Phase: Synthetic Benchmarking for Scalability

We start by comparing the analysis time as the program size (e.g., number of threads) grows. The analysis time of $\text{ILP}_C$ is dependent on the number of iterations required to find a solution and the computation time of each iteration. The former aspect is not so much dependent on the program size but is related to the structure of the program and the distribution of execution costs. Hence, in order to elicit the worst case and best case analysis time of $\text{ILP}_C$ for a given program size, we created two sets of synthetic benchmarks.

![Figure 8: TCCFG template of the synthetic benchmarks.](image)

![Figure 9: Analysis time versus number of threads.](image)
from the four approaches are identical. The analysis time of ILPₚₜₜ model checking and reachability approaches increases exponentially as the number of threads increases. Eventually the analysis time of ILPₚₜₜ and the reachability approach becomes impractical to measure with the benchmarks of greater than 12 threads and 18 threads respectively. As for the model checking approach, it runs out of memory with the 18-threads benchmark. ILPₚₜₜ reaches its worst case performance with this particular structure and cost distribution. Its analysis time is labelled with ILPₚₜₜ in Fig. 9, which more or less mimics the performance of ILPₚₜₜ.

When benchmarking the four approaches against the second set of synthetic TCCFGs, ILPₚₜₜ, model checking and reachability based approaches do not have observable changes in their analysis time. However, the analysis time of ILPₚₜₜ, which is labelled with ILPₚₜₜ in Fig. 9, reduces to less than 0.01 second regardless the increase in the number of threads.

5.3 Second Phase: Evaluation with Real-life Benchmarks

We benchmark the four approaches [4, 11, 13] against a set of real-life PRET-C programs. These programs are taken from [17, 19] and two synthetic programs are added to illustrate the significance of tick alignment. These programs were compiled into PRET-C using the existing tool chain [18]. The details of these programs and benchmark results are shown in Tab. 2. The largest program is WaterMonitor with 3204 lines of code and has 40 concurrent threads.

The WCRTs obtained from all four approaches are identical, which validates the correctness of ILPₚₜₜ. ILPₚₜₜ has the steepest increase in analysis time, and it takes more than an hour for large programs. As for model checking and reachability, the analysis time increases as the state-space grows. Eventually the analysis for large programs, such as RailroadCrossing and WaterMonitor, cannot finish due to long analysis time (> 1 hr) or insufficient memory. For the six small benchmarks (ChannelProtocol to DrillStation), the average analysis time for ILPₚₜₜ, model checking and reachability are 51.2 seconds, 8.06 seconds and 7.62 seconds respectively. None of these three approaches are able to compute the WCRT of larger programs. Reachability [13] provides the closest performance in comparison to ILPₚₜₜ.

In contrast, ILPₚₜₜ is able to compute the WCRTs of such large programs very efficiently. The number of iteration ranges from 1 to 73, and the average analysis time of all the benchmarks is under 2.5 seconds, with peak 5.92 seconds for the Synthetic1 program. The results also show that the analysis time of ILPₚₜₜ is not always affected by program size.

6. DISCUSSIONS

ILPₚₜₜ demonstrates scalable performance in Sec.5.3, thanks to its iterative analysis framework. In this section, we extend our analysis using the benchmark programs presented in Tab. 2, to reveal the relationship between WCRT estimates, analysis time, and number of iterations of ILPₚₜₜ. For all the programs in Tab. 2 that took more than 3 iterations, their WCRT estimates are plotted against the analysis time in Fig. 11. The end of each iteration is marked on the lines.

Fig. 11 shows that the WCRT precision improves with the number of iterations. This trend is best shown by the DrillStation (Fig. 11b) and Synthetic1 (Fig. 11c) programs, with 73 and 37 data points respectively. This is an interesting property as it ties the WCRT precision to the elapsed analysis time. In other words, the ILPₚₜₜ can be interrupted at any time and still able to produce a reasonably precise and safe WCRT estimate. The Synthetic2 program (Fig. 11d) demonstrates a case where the WCRT precision stays the same for iterations 5-7 and 9-11. However, the overall trend of the Synthetic2 program is consistent with the others.

As for the analysis time, there is an extra overhead during the first iteration of ILPₚₜₜ (presented as an offset in the analysis time axis), where the ILPBase is being solved for the first time. This one time overhead may be caused by the initializing process of the ILP solver. After that the analysis time between iterations steadily increases at a slow rate. This may be observed from the Synthetic1 program as the gap between it-

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Threads</th>
<th>WCRT</th>
<th>ILPₚₜₜ iterations</th>
<th>Analysis Time (s)</th>
<th>ILPₚₜₜ</th>
<th>ILPₚₜₜ</th>
<th>Reachability</th>
<th>Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChannelProtocol</td>
<td>591</td>
<td>7</td>
<td>999</td>
<td>3</td>
<td>0.13</td>
<td>5.71</td>
<td>2.55</td>
<td>2.87</td>
<td></td>
</tr>
<tr>
<td>Flasher</td>
<td>816</td>
<td>7</td>
<td>617</td>
<td>2</td>
<td>0.19</td>
<td>11.6</td>
<td>3.22</td>
<td>3.96</td>
<td></td>
</tr>
<tr>
<td>RobotSonar</td>
<td>962</td>
<td>7</td>
<td>1874</td>
<td>1</td>
<td>2.43</td>
<td>14.93</td>
<td>7.06</td>
<td>7.02</td>
<td></td>
</tr>
<tr>
<td>Synthetic1</td>
<td>1291</td>
<td>7</td>
<td>1216</td>
<td>1</td>
<td>5.92</td>
<td>28.16</td>
<td>14.93</td>
<td>14.93</td>
<td></td>
</tr>
<tr>
<td>Synthetic2</td>
<td>1293</td>
<td>7</td>
<td>2114</td>
<td>13</td>
<td>2.25</td>
<td>55.19</td>
<td>17.12</td>
<td>14.9</td>
<td></td>
</tr>
<tr>
<td>DrillStation</td>
<td>1094</td>
<td>15</td>
<td>2751</td>
<td>73</td>
<td>4.38</td>
<td>171.8</td>
<td>3.46</td>
<td>6.05</td>
<td></td>
</tr>
<tr>
<td>CruiseControl</td>
<td>2302</td>
<td>25</td>
<td>1931</td>
<td>5</td>
<td>1.05</td>
<td>&gt; 1 hr</td>
<td>21.19</td>
<td>65.4 &amp; Out of memory</td>
<td></td>
</tr>
<tr>
<td>RailroadCrossing</td>
<td>2713</td>
<td>30</td>
<td>4472</td>
<td>2</td>
<td>1.63</td>
<td>&gt; 1 hr</td>
<td>&gt; 1 hr</td>
<td>68.22 &amp; Out of memory</td>
<td></td>
</tr>
<tr>
<td>WaterMonitor</td>
<td>5204</td>
<td>40</td>
<td>4631</td>
<td>1</td>
<td>3.53</td>
<td>&gt; 1 hr</td>
<td>&gt; 1 hr &amp; Out of memory</td>
<td>75.2 &amp; Out of memory</td>
<td></td>
</tr>
</tbody>
</table>
operations increases. In this benchmark set, ILPc demonstrates the ability of improving the WCRT precision with very small penalty in analysis time.

7. CONCLUSIONS
We have presented a new approach called ILPc specifically targeting the timing analysis of synchronous programs. The key idea of the proposed approach is based on an iterative refinement process that starts with ILP constraints such that the exploration of full state-space is avoided. Through an orthogonal state-space exploration process, we refine the ILP constraints one at a time by exploring only the state-space corresponding to the computed longest tick. Through extensive benchmarking we have shown that the proposed approach provides excellent amortized performance compared to existing approaches. In the future, we will extend the proposed approach to deal with parallel programs, complex memory architectures and processor models.

References


APPENDIX

A. Tick expressions

In this section, we demonstrate tick expressions using TC-CFGs with various structures. A tick expression consists of three elements: the constant, the variable and the min. The TC-CFG in Fig. 12 shows the usage of min. The execution of the join node follows the pattern of T1, but its earliest execution is limited by slowest thread, which is T2 and it takes 4 ticks to complete. Thus for the join node, min is 4.

Fig. 13 shows an example of a thread with two fork nodes in sequence. In this case, the tick expressions of the child threads are generated with respect to the fork node that spawns them. The threads T1 and T2 can never execute together with T3 and T4, and this is enforced by the ILP constraints in ILPmin, hence the tick expressions can be generated separately for the two fork-join pairs.

Finally, Fig. 14 presents tick expressions for a TCCFG with branching and nested loops. As tick expressions are affine functions with lower bound, mathematical techniques can be applied to merge or optimize them. For example, the tick expression for the join node is the union of its predecessors’ tick expressions, which should contains the following three tick expressions:

\[ 3 + 3N + 4N (\text{min} : 3) \]
\[ 4 + 3N + 4N (\text{min} : 4) \]
\[ 6 + 2N + 3N (\text{min} : 6) \]

These tick expressions can be merged and simplified mathematically into ‘6+N (min:6)’, which captures the union of the three sets of numbers of the three tick expressions above.