Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognize the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
Novel algorithms for scalable static analysis of synchronous programs

JiaJie Wang
Oct 2017

Supervisors:
Partha Roop
Enrico Haemmerle, Avinash Malik

Department of Electrical and Computer Engineering
The University of Auckland
New Zealand

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering
Abstract

In modern life, reactive systems are widely used in cyber-physical systems (CPS), such as airplanes and medical devices. One common characteristic of these systems is that they provide services by continuously interacting with our physical world. Thus, they often have strict requirements for functionality and timing. A system is classified as safety-critical if its malfunction may harm our well-being.

Synchronous languages are ideally suited for designing safety-critical reactive systems. These languages provide guarantees on soundness such as determinism and reactivity of correct synchronous programs, which are known to be causal. Consequently, this allows the formal verification of functional properties and Worst-Case Reaction Time (WCRT).

However, since the invention of the synchronous paradigm, there has been not many innovations regarding WCRT analysis. There has been only limited efforts to try and further both precision and scalability during these analyses, especially for programs involving a large number of threads.

In addition, utilising power management in the synchronous paradigm, which is a key aspect for battery powered CPS, has received scant attention. In this thesis, we aim to address these shortcomings.

WCRT analysis is essential for reactive systems, since they interact with our physical world. An output is considered correct if it is delivered in a timely manner. However, as the size of modern systems grow, existing techniques fail to deliver precise WCRT estimates in a scalable manner. Our first attempt to solve this problem is presented in Chapter 3. We propose an iterative WCRT analysis called ILP_c (ILP concurrent), based on Integer Linear Programming (ILP). ILP is conventionally known to be scalable, but produces pessimistic estimates. We discover that this is due to the abstraction of tick alignment in the ILP model, which trades precision for scalability. A key to achieve both precision and scalability is to incorporate the
tick alignment but keep it separate from path analysis. In ILP$_c$ we divide WCRT analysis into two parts, and for each part we develop suitable ILP based techniques. The proposed algorithm combines the two parts in an iterative manner to compute the WCRT.

Our second attempt for scalable WCRT analysis is presented in Chapter 4, and it is based on explicit path enumeration. Conventional explicit path enumeration techniques include model checking and reachability analysis. A well-known problem of this approach is state explosion caused by the composition of concurrent threads. To tackle this problem, we develop a WCRT analysis technique called WCRT algebra, which is an adaptation of a min-max-plus algebra. We propose the idea of WCRT equivalence in modelling the control flow, and subsequently realise this as Tick Cost Automata (TCA). Using TCAs can effectively allow concurrent threads to be quickly composed without sacrificing precision. Both ILP$_c$ and WCRT algebra are benchmarked against the state-of-the-art published WCRT techniques using a set of industrial applications. The results show that both techniques are as precise as the existing techniques while being orders of magnitude faster in many instances. On average, ILP$_c$ is over 10 times faster than published WCRT techniques, and over 1000 times faster for large programs. WCRT algebra is about 3.5 time faster than ILP$_c$.

Finally, the last aspect this thesis tackles is the power management question for synchronous programs. While there are many exiting algorithms available for Real-Time Operating systems, they are not suited to the synchronous paradigm since they are tightly coupled with their adjoining schedulers. In Chapter 5, we propose a framework to combine Dynamic Voltage Frequency Scaling (DVFS) with the synchronous paradigm for the first time. Along with the framework, we develop a bi-criteria optimisation technique to automatically explore the trade-offs between timing and energy consumption using the concept of Pareto Optimality. We evaluate our approach against a conventional approach, where a single frequency is used throughout the execution. The results show that the proposed approach is able to produce more non-dominated options for the user providing more flexibility.

In conclusion, this thesis has pushed the boundary of the synchronous paradigm and opens new opportunities for its applications, especially for safety-critical CPS, which may have energy and timing constraints.
Acknowledgements

This thesis concludes my journey as a PhD student. It is an achievement of not only my own effort, but also the effort of the people around me.

First, I am grateful to my parents, who had provided me their selfless support throughout the degree.

I must thank my main supervisor Partha S. Roop, who lighted my path during the toughest hours, and gave me valuable opportunities to collaborate with world leading researchers. This gratitude extends to my co-supervisor Enrico Haemmerle and Avinash Malik, who provided me with countless suggestions and ideas toward the research.

I would also like to thank the oversea collaborators Alain Girault, Michael Mendler, Bruno Bodin and Sidharta Andalam. To me, you are not only colleagues for the ventures that we had fought through, but friends long after that.

Finally, I would like to thank all the students that I encountered. Matthew Kuo, Mahmood Hikmet, Jin Ro and many more. Thanks for being part of this journey with me. we have been through lots of ups and downs, and the journey would not be complete without any of you.
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 4 is based on the following co-authored work:

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation and writing the paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>75</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Mendler</td>
<td>Problem formulation and editing the paper.</td>
</tr>
<tr>
<td>Bruno Bodin</td>
<td>Problem formulation and editing the paper.</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation and editing the paper.</td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Mendler</td>
<td></td>
<td>06 June 2017</td>
</tr>
<tr>
<td>Bruno Bodin</td>
<td></td>
<td>21 July 2017</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td></td>
<td>21/09/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 3 of the thesis is based on the following co-authored work:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation and writing the paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 5 of the thesis is based on the following co-authored work:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation and writing the paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>75</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alain Girault</td>
<td>Problem formulation and editing the paper.</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation and editing the paper.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alain Girault</td>
<td>[Signature]</td>
<td>12/05/2017</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>[Signature]</td>
<td>21/09/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
## Contents

1 Introduction  ........................................... 1
   1.1 Literature Review ...................................... 3
      1.1.1 Synchronous programming ........................... 3
      1.1.2 Timing analysis ..................................... 5
      1.1.3 Power management .................................. 11
   1.2 Problem statement .................................... 13
   1.3 Contributions ......................................... 13
   1.4 Thesis organisation .................................... 13
   1.5 Publications and thesis formatting .................... 14

2 Background ............................................. 15
   2.1 Precision Timed C ...................................... 15
   2.2 Timed Concurrent Control Flow Graph .................. 16
   2.3 Implicit Path Enumeration Technique .................... 18

3 Scalable WCRT analysis using ILP ......................... 21
   3.1 Motivating example ..................................... 22
      3.1.1 Implicit path enumeration ........................... 24
3.1.2 Explicit path enumeration ........................................ 24
3.1.3 The proposed approach ......................................... 25

3.2 TCCFG running example .......................................... 27

3.3 The proposed technique ILP\(_C\) .................................... 27
3.3.1 The ILP model ILP\(_{base}\) ........................................ 28
3.3.2 Tick expressions .................................................. 37
3.3.3 Verifying an execution path for tick alignment ................. 39
3.3.4 Eliminating the infeasible EOT combination .................... 43

3.4 Results ........................................................................ 44
3.4.1 The benchmarking settings .................................... 44
3.4.2 Synthetic benchmarking for scalability ....................... 45
3.4.3 Realistic benchmarks ............................................. 47
3.4.4 Analysis time and precision trade-off ....................... 49
3.4.5 WCRT versus iteration in ILP\(_C\) ................................. 49

3.5 Conclusions ............................................................. 52

4 Timing analysis of synchronous programs using WCRT Algebra 53
4.1 Motivation ............................................................... 54
4.2 Running example ..................................................... 59
4.3 The WCRT Algebra .................................................. 59
4.3.1 Mathematical structure .......................................... 59
4.3.2 Tick Cost Automata (TCAs) ................................... 61
4.3.3 Worst Case Reaction Time Analysis ......................... 62
4.3.4 Modeling the TCA of a thread ........................................... 62
4.3.5 Modeling the TCA of a fork box ........................................ 68
4.3.6 Modeling the TCA of an abort box ................................. 69

4.4 Implementations ............................................................... 70
4.4.1 The Worst-Case Reaction Time (WCRT) analysis logic .......... 71
4.4.2 Part2: Converting a Tick-Cost Automaton (TCA) to a WCRT equivalent graph ........................................... 72
4.4.3 Part2: Generating the TCA for a box ................................. 73

4.5 Evaluation ................................................................. 74
4.5.1 Benchmark settings ..................................................... 75
4.5.2 Existing benchmark programs ...................................... 75
4.5.3 Theoretical properties .............................................. 76

4.6 Conclusions ................................................................. 80

5 Synchronous paradigm with DVFS ........................................ 81

5.1 Preliminaries ............................................................. 83
5.1.1 Running example ...................................................... 83
5.1.2 Worst-Case Energy Consumption .............................. 84
5.1.3 Dynamic Voltage and Frequency Scaling ...................... 84
5.1.4 Pareto Optima ....................................................... 85

5.2 DVFS schemes for synchronous programs ......................... 86
5.2.1 The running example with Dynamic Voltage and Frequency Scaling (DVFS) scheme ........................................... 87
5.2.2 Comparing DVFS schemes ........................................ 88
List of Figures

1.1 The tick by tick execution of a synchronous program, and the composition of a reaction. .................................................. 3

1.2 The execution time of a program during its execution. If the estimated WCET is smaller than the deadline, the actual WCET will also be smaller than the deadline. ............................................................... 6

1.3 Overview of static Worst-Case Execution Time (WCET) analysis. .............. 7

1.4 Overview of the synchronous design flow and WCRT analysis. ................. 8

1.5 Two possible frameworks for the hybrid approach. .............................. 10

2.1 A Precision Timed C (PRET-C) example. ........................................ 16

2.2 A Timed Concurrent Control Flow Graph (TCCFG) running example. Used for demonstrating the formulations of WCRT algebra. The TCCFG is divided into boxes based on the hierarchy. .................................................. 17

2.3 An example of SCFG for illustrating Implicit Path Enumeration Technique (IPET). 19

3.1 FSM representation of a three-thread synchronous program. ................. 23

3.2 Execution trace of the three-threaded FSM. ..................................... 23

3.3 A running example of TCCFG. ....................................................... 26

3.4 The overview of the proposed WCRT analysis technique ILP_C. ............ 27

3.5 Thread hierarchy of the running example. ........................................ 31

3.6 The structure of preemption in TCCFG. ........................................... 35

xiii
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7</td>
<td>The running example with tick expressions. The computation and condition</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>nodes are abstracted.</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Overview of the verification process.</td>
<td>40</td>
</tr>
<tr>
<td>3.9</td>
<td>TCCFG template of the synthetic benchmarks.</td>
<td>45</td>
</tr>
<tr>
<td>3.10</td>
<td>Analysis time versus number of threads.</td>
<td>46</td>
</tr>
<tr>
<td>3.11</td>
<td>WCRT vs accumulated analysis time for the benchmark programs</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Motivating example for comparing the proposed and the existing techniques.</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Existing technique flatten the functional composition to obtain a single</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>automaton.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Converting $T_A$ and $T_B$ into TCAs, and composing them to compute the WCRT.</td>
<td>57</td>
</tr>
<tr>
<td>4.4</td>
<td>A TCCFG running example. Used for demonstrating the formulations of WCRT</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>algebra. The TCCFG is divided into boxes based on the hierarchy.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>The general form of a TCA.</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>TCCFG section $Box_{fork}$: concurrent threads $T1$ and $T2$ with virtual</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>start, virtual end and EOT nodes.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>TCCFG section $Box_{abort}$: the check abort threads $CheckA$ and abort body</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>$ABody$ with virtual start and virtual end nodes. The box $Box_{fork}$ is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>replaced with a WCRT equivalent graph of $TCA(Box_{fork})$ indicated by the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dashed box.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Converting a TCA in to a sequential TCCFG.</td>
<td>72</td>
</tr>
<tr>
<td>4.9</td>
<td>Overview of how to construct the TCA for a given box.</td>
<td>73</td>
</tr>
<tr>
<td>4.10</td>
<td>Python code example for the computation node $B3$.</td>
<td>74</td>
</tr>
<tr>
<td>4.11</td>
<td>Synthetic A.</td>
<td>77</td>
</tr>
<tr>
<td>4.12</td>
<td>Synthetic B.</td>
<td>78</td>
</tr>
<tr>
<td>4.13</td>
<td>Results of the model checking, ILP$_C$ and timing simulation in WCRT algebra</td>
<td>79</td>
</tr>
</tbody>
</table>
4.14 Results of the model checking, ILP_C and timing simulation in WCRT algebra with Synthetic B. ......................................................... 79

5.1 The PRET-C code of the running example. ........................................ 83

5.2 The TCCFG of the running example. .................................................. 83

5.3 An illustration of Pareto optima in the (WCRT, WCEC) space. ................. 86

5.4 The PRET-C running example with frequency annotations. .................. 87

5.5 The Pareto front of the running example. .......................................... 89

5.6 There are a few known Pareto optima in the (WCRT, WCEC) space. These optima have all the DVFS control points set to a single frequency. ............. 91

5.7 Four searching algorithms for constructing the Pareto front in the (WCRT, WCEC) space. ............................................................. 92

5.8 Given a constraint and the knowledge of the Pareto optima with fixed frequency, we can determine the starting point of the search and the maximum/minimum frequency that should be used. .......................... 93

5.9 Details of backward searching with WCRT constraint. ......................... 93

5.10 Converting the TCCFG in Fig. 5.2 into four virtual TCCFGs for emulating DVFS. 97

5.11 The DVFS algorithm for the backward search with a WCRT constraint. .... 102

5.12 The estimated Pareto fronts of the benchmark programs. Turing WCRT into constraints. ............................................................. 105

5.13 The estimated Pareto fronts of the benchmark programs. Turing WCEC into constraints. ............................................................. 106

5.14 Merging all the estimated Pareto fronts and compared with two existing techniques. ............................................................. 107
# List of Tables

1.1 Chapters and publications. ........................................... 14

2.1 Tick snapshots of the PRET-C example, with execution cost for each tick. .... 18

3.1 Truth table for join node with respect to one child thread .................. 32

3.2 Truth table for Join1 ...................................................... 33

3.3 Analysis time summary .................................................. 48

3.4 Precision trade-off ....................................................... 50

4.1 Benchmarking results for model checking (MC), ILP$C$ and WCRT algebra (WA). 76

5.1 The available operating points on our target processor and their power consumption. .................................................. 85

5.2 The energy consumption per clock cycle of the target processor (Unit: µJ). .. 85

5.3 The execution time of the computation nodes in Fig. 5.2 at four different frequencies (Unit: µs). ........................................... 88

5.4 The energy consumption of the computation nodes in Fig. 5.2 at four different frequencies (Unit: µJ). ........................................... 88

5.5 DVFS scheme examples for the running example. .............................. 89

5.6 Information of the benchmark programs. ..................................... 103
5.7 Analysis time for estimating the Pareto fronts of the benchmark programs. 
(Unit: seconds) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
The 1970s set an important landmark in human history. It was the beginning of the digital revolution, which forever changed the way we live. The most notable change was the rapid adoption of embedded systems, such as Automated Teller Machines (ATMs), traffic lights, and calculators. All these machines and devices are controlled by embedded systems, and they have become a part of our daily life. In addition to providing us unprecedented convenience, embedded systems also empower us to create what was previously not possible, such as advanced deep-sea robots, intelligent interplanetary explorers, smart televisions in our homes, automated factories, and many other examples besides. It is fair to say that a big part of our modern world is built based on the applications of embedded systems. Because these systems constantly interact with our physical world, they are also known as reactive systems [1].

Reactive systems are difficult to design because their operations involve interaction with the physical world. This is unlike general purpose computer systems where everything is encapsulated within the digital domain. Some reactive systems can potentially harm our well-being if they malfunction. Typical examples of these include implantable pacemakers, Anti-lock Braking Systems (ABS) in automobiles and the autopilot system in aeroplanes. These systems are classified as safety-critical [2]. For design of safety critical systems, international standards and regulations are put in place to enforce good engineering practice. For example, medical devices must comply with the IEC60601 standard [3] if intended for commercial use. Similarly, automobile systems are subject to the ISO26262 standard [4]. Certifying for compliance requires evidence of safety and reliability, that is, proving the risk of endangering our well-being is at an acceptably low level.
Synchronous languages were specifically invented for tackling the design challenge inherent in reactive systems. They are high-level specifications, which support concurrency and preemption, allowing most reactive algorithms to be naturally expressed. In addition, the synchronous paradigm is developed based on formal semantics which ensures that the execution of the system is deterministic [5]. As a result, the functionalities and timing of synchronous programs can be formally verified. In other words, a system developed using synchronous languages can be mathematically proven to always respond correctly in a timely manner. This is a desired feature when designing safety critical systems as formal verification can produce strong evidence for safety that aids in the certification process. Thus, the synchronous paradigm is considered a top choice for safety critical systems.

However, the synchronous paradigm was proposed around 25 years ago [5, 6] and many of its tools and languages have not been actively developed since then. Hence, many of its associated analysis techniques and language features are starting to be dated, particularly those in the area of timing analysis and power management. Modern systems, which have been growing in size to accommodate ever expanding features and functionalities, are beginning to exceed the capabilities of existing timing analysis techniques. Synchronous program sizes once considered abnormal are now considered typical, and analysis of these programs cannot be completed within practical time frames. As a result, designers have to rely on abstraction to improve the scalability, but this abstraction comes at a cost: large overestimates for the timing bounds. This, at times, can unnecessarily increase the end product cost due to the use of more powerful platforms to meet the timing deadline.

Another drawback of the paradigm is its lack of direct support for processor power management. Power saving technologies can be seen in almost all modern processors, and it is essential for portable battery operated devices. Many algorithms exist for Real-Time Operating Systems (RTOSes) to utilise these technologies; however, no work exists for the synchronous paradigm because of its bare-metal approach, that is, executing directly without a scheduler. Many safety critical applications have power constraints, and they can benefit from using these power saving technologies.

In this thesis, we intend to address these two shortcomings of the synchronous design approach. This work was done based on the synchronous language Precision Timed C (PRET-C) [7, 8], and targets single-core processors. We address the scalability problem of timing analysis by studying its in-depth causes, and developing new theories and two analysis techniques that scale well with large synchronous programs. For power management, we extended the synchronous paradigm to integrate with the power saving technology Dynamic Voltage and Frequency Scaling (DVFS), and developed a bi-criteria optimisation technique to visualise the timing and energy trade-offs.
1.1 Literature Review

1.1.1 Synchronous programming

The synchronous paradigm was invented for designing reactive systems. Because reactive systems have to continuously interact with the physical world, they usually have strict system requirements: both functional and non-functional. Functional requirements can be that the system must exhibit a repeatable and predictable behaviour (deterministic), must be responsive (liveness), and always produce the correct response. For non-functional requirements, an example can be the Worst-Case Reaction Time (WCRT) of the system, i.e., the longest time for a system to response to an event. Given most reactive algorithms are highly concurrent in nature, it is very difficult to guarantee the satisfaction of these requirements [9].

The synchronous paradigm eases this design challenge, and allows programmers to focus more on the design of the system. The essential constructs for designing reactive systems, such as concurrency and preemption, are supported in most synchronous languages. On top of that, the execution of synchronous programs is based on formal semantics, which facilitate verification of system properties, ensuring that system requirements are met.

![Diagram](image)

The central tenet of the synchronous paradigm is to drive the execution of a program using a notion of logical time, called ticks. Hypothetically, the execution of a synchronous program is a sequence of ticks, and the boundaries of ticks are marked in the source code by the programmer using Pause/EOT (End Of Tick) statements. Fig. 1.1 shows an illustration of this tick by tick execution. In each tick, the program first samples and buffers the inputs from the environment, then the computation takes place (i.e., concurrent threads execute), and the outputs are emitted together at the end. This completes one reaction to the environment.

This notion of a reaction simplifies the verification for timing. Since a synchronous program performs one complete reaction in each tick, the longest response time of the system is bounded by the longest execution time of a tick. This time is known as the Worst-Case Reaction Time (WCRT) of the system. The synchrony hypothesis states that the reaction of a system appears to be instantaneous to its environment if the WCRT is shorter than the minimum interarrival time of the events [5, 6].

In addition to employing the logical tick, synchronous languages also do not allow recursions and instantaneous loops (i.e., all paths in a loop must contain at least one pause or EOT
statement, and hence each iteration takes more than one tick to complete.) Thus, the execution time of each reaction is always bounded. This gives synchronous programs very simple and clear execution semantics.

**Types of synchronous languages:** The synchronous design approach is proven to be successful, and it has become essential in the design of safety critical systems in a wide range of domains [10,11,12]. To tailor the needs of these different domains, synchronous languages come in various forms. There are classical textual synchronous languages such as Lustre [13, 14], Esterel [5], and Signal [15,16]. There are also visual synchronous languages that are based on diagrams, such as SCADE [12,17] and SyncCharts [18]. Each synchronous language carries a different feature set that is suitable for different applications. For example, Esterel is suitable for designing control-dominated systems, and Lustre is suitable for data-dominated systems.

In addition to the domain-specific synchronous languages, there are also synchronous languages developed based on the C language. Some of the most noticeable examples include ECL [19], PRET-C [7, 8], and Synchronous C [20]. The intention of C-based synchronous languages is to allow seamless intergeneration with other C programs and allow reuse of existing C libraries. Since the C language is the language of choices for embedded system design.

Synchronous languages can also be used to design distributed systems. This is achieved by extending the synchronous paradigm to a Globally Asynchronous, locally Synchronous (GALS) model. This model consists of two layers: threads in the same nodes execute synchronously like a normal synchronous program, but each node is asynchronous to each other. Communication between nodes is achieved using convention message passing. Languages that are based on GALS include FunLoft [21] and SystemJ [22].

**Executing synchronous programs:** The first approach is to compile it to a single-core general-purpose processor. This approach often uses the C language as an intermediate media in the process, as it is widely supported across embedded platforms. Given a synchronous program, it is first compiled into sequential C code, where concurrency and preemptions are “compiled away” [31,32]. Then the C code is compiled into a binary and deployed on the target processor. Alternatively, synchronous program can also be compiled into tasks and then use a RTOS to schedule their execution on the processor [33, 34]. Of these two methods, compiling into sequential C code is better when it comes to static analysis of the program, as the code determines the precise interleaving of threads at compile time. In contrast, RTOSes introduce overhead and dynamicity, which can make execution less predictable and more difficult to analyse.

There are three approaches to execution of a synchronous program. The second approach to use a special class of processors called reactive processors [23,24,25,26,27]. Each reactive processor are specifically designed for a single synchronous language, and have special datapath and control units that allow some language features, such as preemption and concurrent execution, to be directly mapped to instructions. As a result, WCRT analysis on such a processor is relatively simple [28,29]. This is the most efficient approach to execute synchronous programs. Concurrency is handled using single-core multi-threading [26,27], or multi-processors [30].
However, at this stage, reactive processors are not ready for practical use as they are still in the research domain, and offer no off-the-shelf products.

Lastly, synchronous programs can execute on a multi-core or multi-processor platform. The common approach here is to fragment the sequentialised program, and parallelising the fragments while preserving the execution semantics [35, 36]. However, not all synchronous programs can benefit from using a multi-core system, since the data dependencies inherent in synchronous programs can sometimes demand sequentialised execution to ensure determinism. To overcome this problem, a recent synchronous language called Fore-C was developed [37]. Fore-C achieves a more consistent parallelism by having a separate copy of the shared variables for each thread. Concurrent threads can execute at their own pace, accessing only their local copies of the variables, until the end of a tick where the all the local copies of shared variables are merged and updated. The limitation of this approach is the operation to merge the variable has to be commutative and associative (e.g., plus and minimum) in order for a deterministic result.

Out of the three approaches, compiling to sequential C code for a single-core processor is by far the most established and utilised approach. In this thesis, we will focus on this approach.

### 1.1.2 Timing analysis

Because reactive systems are constantly interacting with the physical world, their actions are usually time sensitive. An output is only correct if it is produced before its deadline. Missing a deadline is considered a malfunction. Therefore, these time sensitive reactive systems are known as real-time systems, and timing analysis must be performed to ensure they can meet their required deadlines.

#### Overview of WCET analysis

Timing analysis is widely known as Worst-Case Execution Time (WCET) analysis, which is computing the longest execution time between two points in a program [38, 39].

Computing the exact WCET is difficult. Fig. 1.2 shows an example of probability distribution of a program’s execution times. The execution time of a program varies depending on the hardware state, the program’s state, and its inputs. The combination of these three factors produces an extremely large search space, which is computationally impractical to analyse. Thus, conventionally, we compute an upper bound for the WCET instead. WCET analysis computes an overestimate of the actual WCET. If the estimated WCET satisfies the deadline, the actual WCET will also satisfy the deadline (i.e., actual WCET ≤ estimated WCET ≤ deadline).

In this thesis, we measure the performance of a timing analysis technique in terms of precision and scalability. The precision of a technique refers to how close is the computed WCET (i.e.,
estimated WCET) compares to the actual WCET. The closer to the actual WCET, the more precise is the technique. In other words, knowing the estimated WCET is a sound overestimate, the smaller value, the better. The scalability of a technique is referring to the rate of increasing in analysis time as the program grows in size. Generally, precision and scalability are two antagonistic qualities. Techniques that are precise usually involve more computation and hence are less scalable. Vice versa, scalable techniques are usually less precise.

There are two approaches for WCET analysis in general [39]: dynamic and static. Dynamic analysis is also known as the measurement-based timing analysis, which runs the program on the actual hardware and excites it with a set of synthetic inputs. The timing is obtained by externally observing the system. This approach is very simple and efficient as there is no modelling or abstraction involved. However, it has one critical drawback: it often underestimates the WCET because of insufficient coverage of the inputs. For this reason, dynamic analysis is only practical for evaluating average performance or other non-critical aspects, and has rarely been used to estimate the upper bound for timing in safety critical applications.

In contrast, static timing analysis always computes an overestimate of the WCET, and thus it is always safe to use. This approach abstracts the hardware and program logic as mathematical models, and analyses the models for timing with consideration of all possible inputs. There is no actual execution involved in the process.

In this thesis, we consider only static timing analysis (referred to as timing analysis in the remainder of this thesis), targeting single-core processors. As this is the most established approach for the synchronous design flow (See Sec. 1.1.1).
1.1 Literature Review

Figure 1.3: Overview of static WCET analysis.

**Static timing analysis**: Fig. 1.3 shows an overview of static timing analysis. Timing analysis operates alongside the design flow, determining whether a given binary can meet its deadline when executing on its target hardware. Timing analysis takes place in two steps. First is to construct a Control Flow Graph (CFG) from the binary, and annotate the nodes in the CFG (i.e., blocks of instructions) with execution cost (i.e., 1). This step involves modelling the speculations of hardware (e.g., pipeline [40] and memory [41]) and estimating the execution cost of each instruction in terms of processor cycles. On top of that, the CFG can optionally include context from the source code to assist in pruning infeasible paths. This can include variable assignment, computation, conditions of the branches etc. All in all, the CFG is an abstracted representation of the system, containing both the program logic and the timing information of the underlying hardware. OTAWA [42] and Chronos [43] are the two most notable tools used for constructing CFG in the public domain.

Once the CFG is produced, the second step is to compute the WCET from the CFG (i.e., 2). This is where the precision and scalability trade-off occurs. There are two main approaches: implicit path enumeration and explicit path enumeration.

Implicit path enumeration is the most well-known approach for timing analysis [44]. It formulates the CFG as an Integer Linear Programming (ILP) problem, and in the process abstracts away the explicit execution order of the program [45]. For example, an execution path is considered as a set of nodes and edges in this approach, instead of a sequence of nodes and edges (i.e., ordered nodes and edges). This abstraction results in great scalability, and thus this approach is used by many of today’s commercial and research tools such as aiT [46], Bound-T [47] and SWEET [48]. The conventional implementation of this approach is known by a very similar name, called Implicit Path Enumeration Technique (IPET) [49]. While the approach is scalable, it is less precise than its counterpart explicit path enumeration, due to its limitations on expressiveness [50] and inability to detect infeasible execution paths during the analysis.

Explicit path enumeration is a more complex approach [51,52]. In contrast to implicit path enu-
meration, explicit path enumeration preserves the complete execution semantics. The idea is to create a complete formal model from the CFG, and use it to emulate the program execution. Compared to implicit path enumeration, the biggest advantage of explicit path enumeration is the enhanced expressiveness [53], which allows the techniques to more accurately model the execution of the program, and thereby computing a more precise estimate. However, this accurate modelling also induces serious state explosion problems [54, 55]. Thus, techniques based on this approach are generally only practical for small programs.

Overview of WCRT analysis

As previously mentioned, timing analysis of a synchronous programs is essentially computing the longest execution time of a tick, which is known as WCRT analysis. WCRT analysis can be considered as a subset of WCET analysis. However, there are a few differences making conventional WCET analysis techniques unsuitable for WCRT analysis.

Firstly, loops are less concerning in WCRT analysis, since the synchronous paradigm does not allow instantaneous loops. This simplifies the analysis for WCRT. However, conventional WCET analysis techniques do not take advantage of this. Second, conventional WCET analysis techniques were developed for sequential programs, thus they solely rely on the context of the sequential C code to prune infeasible paths. This is not efficient in analysing WCRT due to the highly concurrent natural of synchronous programs.

![Figure 1.4: Overview of the synchronous design flow and WCRT analysis.](image)

Fig. 1.4 shows the overview of the synchronous design flow and the WCRT analysis process. Conventionally, the CFG is constructed based on the context of the sequential C code, since the C code is usually the source code for sequential program. However, in the synchronous design flow, the sequential C code is compiler generated, therefore it does not reflect the intention of the original design. It is better to use context of the synchronous program instead. This
1.1 Literature Review

will expose the high-level control flow, such as concurrency and preemption, for the WCRT analysis, making the CFG a more accurate representation of the system.

This idea of exposing high-level context for used in WCRT was first explored in 2008 [56]. Since then, a few works have followed up on this idea [57, 58, 59, 60]. These works present dedicate techniques for timing analysis of synchronous programs, and they will be the main references for comparison in this thesis. In the following subsections, we present our systematic study of these existing WCRT analysis techniques, and highlight their strengths and weakness. We categorise all the existing WCRT analysis techniques into three major approaches, namely implicit path enumeration, explicit path enumeration, and the hybrid of the two.

**Implicit path enumeration:** In [56], high-level context was used in WCRT analysis for the first time. It is based on ILP and uses the high-level context to prunes simple conflicting pairs in the CFG. The precision improvement is limited and leaves much to be desired. Thus, the work was subsequently improved using a hybrid approach in [57] and [61]. We will discuss these related works later under the Hybrid approach section.

**Explicit path enumeration:** There are two notable works based on this approach. In [58], a model checking based WCRT analysis was proposed. This technique makes use of the high-level context, e.g., the preemption that has been “compiled away”, to model the CFG as a set of interacting finite state automata that are synchronously composed together. Then, the WCRT computation is formulated as a series of properties to be checked by the model checker. Alternatively, a reachability based technique was proposed in [59], which also utilises the high-level context to create custom concurrent automata from the CFG. The WCRT is computed by performing reachability tests on automata while applying optimisation on the fly. Both techniques have the same level of precision, with the reachability-based technique being marginally more scalable.

**Hybrid approach:** Since neither implicit nor explicit path enumeration are ideal, efforts have been made to combine the two approaches in an attempt to achieve a balance. The essence is to improve the precision of implicit path enumeration by removing infeasible paths through addition of more ILP constraints. The idea of the hybrid approach is to start with a baseline ILP model (i.e., implicit path enumeration), and then use explicit path enumeration techniques to generate additional ILP constraints to improve its precision.

There are two frameworks for combining implicit and explicit path enumeration, namely the single pass framework, and the iterative framework. The single pass framework (Fig. 1.5(a)) is used by [57] and [61], which first fully generate the additional ILP constraints by enumerating a heavily abstracted automaton, then add the ILP constraints to the baseline ILP model and solves the combined model to compute the WCRT. The abstracted automaton used here is much simpler than a full-blown model checking or reachability model [58, 59], thus it takes much less time to enumerate.

The iterative framework, on the other hand, generates the ILP constraints on the fly during the analysis (Fig. 1.5(b)). This framework is used by [60], where the baseline ILP model is first
solved to quickly obtain a WCRT estimate and its associated critical path, then the path is verified using a model checker. Here, the model checker uses a full-blown model to deliver rigorous results. If the critical path is found to be infeasible, indicating the WCRT estimate is invalid, a new ILP constraint is generated and added to the baseline ILP model and then the analysis starts over again. This iterative process continues until the verification is successful. In Fig. 1.5(b), this process is highlighted with bold arrows.

Both frameworks have their strengths. The single pass framework uses a heavily abstracted automaton, which is faster to evaluate than the full-blown automaton used in the iterative framework. Moreover, the single pass framework only evaluates the automaton once, but the iterative framework has to evaluate the automaton in every iteration. On the other hand, the iterative framework can be more efficient in the sense that the ILP model is being gradually refined. Its size (i.e., number of ILP constraints) will only grow as large as it needs to be.
However, in reality, both frameworks do not scale. As more exclusions (i.e., ILP constraints) are added, the solving time of the ILP model increases dramatically \cite{54, 60}, to a point where using the hybrid approach scales worse than just using an explicit path enumeration technique alone (e.g., model checking).

**Conclusions**

WCRT analysis is a well-explored research topic. The three approaches for analysing CFGs, implicit and explicit path enumeration, as well as their hybrid variants, have been extensively studied. Each approach has its own strengths, and the question of which one is better remains undecided amongst researchers \cite{53, 54, 55}. Choosing one technique over another is simply choosing a trade-off between precision and scalability. However, all existing studies have been focusing on the accuracy of modelling, and have overlooked one crucial factor: the efficiency. When solving the model, how much of the computation is actually contributing toward WCRT precision? We hypothesise that a large portion of the computation is unnecessary. In this thesis, we attempt to understand the factors that cause the state explosion problem and evaluate their necessity in the analysis; then exploit the potential of new techniques which are both precise and scalable.

**1.1.3 Power management**

As more and more reactive systems become battery operated, power management has caught increasing attention in the design of modern systems. The principle of power management is simple, which is to slow down or turn off the processor, or on-board components, to conserve energy. Noticeable power saving technologies include power gating, clock gating and Dynamic Voltage and Frequency Scaling (DVFS). These technologies are tricky to use in real-time systems, since they cause penalties in the execution time when active. Hence, the power management algorithms for real-time systems must take into account timing deadlines while attempting to save as much energy as possible. To the best of our knowledge, there is no such algorithm for the synchronous paradigm. We will attempt to address this shortcoming by combining the paradigm with DVFS. In this subsection, we will present a brief literature review on the DVFS algorithms for Real-Time Operating System (RTOS).

A processor capable of DVFS may operate at several discrete frequencies. To ensure correct processor operation, each frequency is accompanied by a voltage setting. Thus a processor has several operation modes, each one being a pair (voltage, frequency). A point during program execution, where the processor changes its operation mode, is referred to as a *DVFS control point*. DVFS can be controlled through software to optimise the performance and energy consumption, and find good trade-offs by changing the operation mode dynamically. For a real-time system, the energy consumption can be safely reduced by consuming the slack (i.e., the time between program completion and the deadline) in exchange for a lower processor frequency.
Existing DVFS algorithms for Real-Time Operating Systems

There are many DVFS algorithms available for RTOS. A DVFS algorithm needs to make two decisions: the control point locations in the program, and the frequency value associated with each control point. Based how a DVFS algorithm makes these two decisions, it may be broadly classified as *online, offline, or hybrid*.

Online [62,63,64] algorithms decide both the control point locations and the frequency values during runtime over a RTOS. These algorithms are closely associated with the underlying scheduling policy and hence are scheduler dependent. Offline algorithms [65,66], on the other hand, make both decisions at compile time. Hybrid algorithms [67, 68] make a combination of offline and online decisions: DVFS control points are inserted into the program at compile time, but the frequency values of the control points are computed during runtime.

Both the online and hybrid approaches are best effort techniques, in the sense that they rely on extracting the runtime information, such as the current slack time, to compute the next processor frequency. Because of the dynamicity in runtime information, techniques based on these two approaches can only be evaluated using simulation, which may yield variable results depending on the simulation and benchmark settings. This uncertainty makes these two approach less favourable for safety-critical applications. On the other hand, offline algorithms allow static analysis (e.g., estimating the worst case battery life or peak power). The existing techniques for computing the locations and frequencies offline [65,66] are based on profiling, i.e., optimising for the most frequently executed path. This approach greatly simplifies the optimisation problem, from ensuring *all possible execution paths* can meet the deadline, to ensuring only *a single execution path*. As a result, the produced scheme may violate the deadline on non-profiled paths.

Conclusions

While using DVFS in a real-time system is a well-researched topic, synchronous programs have not benefited from the existing studies. The main reason is that all existing DVFS algorithms are tightly coupled with the scheduler of a particular RTOS, however, synchronous programs are directly compiled into sequential C code (i.e., with no scheduler). This leaves a blank page for the synchronous paradigm, and in this thesis, we will explore the potential of combining the synchronous paradigm and DVFS.
1.2 Problem statement

The synchronous paradigm is a top choice for designing safety-critical systems because of its unique advantage when it comes to formal verification. However, considering its design flow and features with today’s standard, it is falling short when it comes to scalable WCRT analysis and power management. The goal of this thesis is to address these shortcomings.

The objective of this goal is two fold, that is to (1) develop new scalable and precise techniques for WCRT analysis, and (2) develop a suitable framework to combine DVFS with the synchronous paradigm.

1.3 Contributions

The main contributions of this thesis are summarised below:

- We systematically study, categorise and compare the existing techniques to identify the in-depth causes of the state explosion problem, and based on the findings, we develop two new WCRT analysis techniques, ILP\textsubscript{C} and WCRT algebra. Against the common belief, these two techniques are demonstrated as both precise and scalable. In our benchmark, they are an order of magnitude faster than all published state of the art techniques while being as precise, with WCRT algebra being faster than ILP\textsubscript{C}. Because the two proposed techniques are developed to specifically avoid the common causes of the state explosion problem, they exhibit no tendency of this issue during our benchmarking.

- We propose a DVFS scheme for the synchronous paradigm, and develop a bi-criteria optimisation technique to estimate the Pareto front in the “timing versus energy consumption” space. In our experiment, the proposed analysis technique is able to generate a set of settings with various timing and energy consumption trade-offs. Compared with the conventional approach where only a single processor frequency is used throughout the execution, our approach is much more useful in the sense that it provides more options (i.e., trade-offs) for the user to choose from. To the best of our knowledge, this is the first attempt to introduce power management for the synchronous paradigm.

- In this thesis we develop new tool chains in the area of WCRT analysis and power management for the synchronous design approach, which makes the paradigm better suited for designing modern systems.

1.4 Thesis organisation

This thesis consists of five remaining chapters. All the work in this thesis is based on the synchronous language PRET-C and its intermediate format Timed Concurrent Control Flow
Graph (TCCFG). In Chapter 2, we present some background on this language and its intermediate format. Chapters 3-5 are comprised of three publications. In Chapter 3, we present the WCRT analysis technique $\text{ILP}_\text{C}$, and subsequently in Chapter 4 we present the WCRT analysis technique WCRT algebra. In Chapter 5 we present the proposed DVFS scheme for the synchronous paradigm, and the associated bi-criteria optimisation technique. Lastly, the thesis is concluded in Chapter 6.

### 1.5 Publications and thesis formatting

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>J. J. Wang, P. S. Roop and M. Kuo, <em>Can Embedded Systems be at the Heart of New Zealand’s Knowledge Economy?</em>, Design Automation Conference Global Forum (DAC-GF), 2013</td>
</tr>
</tbody>
</table>

Table 1.1: Chapters and publications.

The publications made during the degree are summarised in 1.1. Chapters 3-5 were written based on some of these publications, but have been modified to improve consistency, and significantly expanded to reflect the work after the publications and to accommodate the content which was stripped due to publication page limit.
The synchronous paradigm is the central subject of this thesis, and we choose to use the synchronous language Precision Timed C (PRET-C) [7, 8] for this research. In this chapter, we present an overview of the PRET-C language and its intermediate format Timed Concurrent Control Flow Graph (TCCFG). To make the thesis self-contained, we also include a brief introduction to the Implicit Path Enumeration Technique (IPET).

2.1 Precision Timed C

In the embedded systems design space, the C language is the industry standard, and PRET-C provides it a synchronous extension. This approach gives PRET-C a few edges over the other synchronous languages. First, it allows existing C libraries to be seamlessly used in PRET-C programs. Second, compiling to sequential C code can be implemented using macros, making the generated C code lightweight and ideal for resource constrained systems. Third, the macro based implementation also facilitates Worst-Case Reaction Time (WCRT) analysis and provides good traceability between the binary and the synchronous program’s source code.

An example of a PRET-C program is shown in Fig. 2.1. PRET-C extends the C language with three major synchronous statements: The End Of Tick (EOT) statement, which marks tick boundaries; the PAR statement for spawning concurrent threads; and the abort statement for preemption. An abort consists of two parts, the abort condition validation (e.g., line 17), and the abort body (e.g., lines 14-16). The abort body is preempted when the abort condition is true. There are two types of aborts: strong and weak. In a strong abort, the abort condition is checked before the execution of the abort body; and in a weak abort, the abort condition is
1 ReactiveInput(int, In1, 0);
2 void T1(){
3     EOT;
4     foo_B8();
5 }
6 void T2(){
7     EOT;
8     foo_B10();
9     EOT;
10 }
11 void main(){
12     // A strong abort
13     abort{
14         EOT;
15     }
16     // Spawning T1 and T2
17     PAR(T1, T2);
18 }
19     foo_B16();
20 }

Figure 2.1: A PRET-C example.

checked after the execution of the abort body. This is akin to the difference between a while {...} loop and a do{...} while loop.

The main thread in this example contains a strong abort. When the environment input In1 is true, everything enclosed inside the abort body is preempted immediately. The abort itself terminates when both T1 and T2 terminate (i.e., join) or when the preemption In1 takes place.

PRET-C does not have signals like most of the other synchronous languages, thus it relies on executing concurrent threads in a static order to ensure determinism. This priority based execution will be illustrated in the next section.

2.2 Timed Concurrent Control Flow Graph

The execution of a PRET-C program can be entirely represented using a TCCFG. Thanks to its macro based implementation, the tool chain for PRET-C is able to construct a TCCFG directly from the binary by correlating it with the high level context from the PRET-C program.

The TCCFG in Fig. 2.2 reflects the control flow of the PRET-C program in Fig. 2.1, with execution cost annotated beside each node. A TCCFG has the following types of nodes: the conventional start, end, computation and condition nodes, with additional abort-start and abort-end nodes for preemption, fork and join nodes for concurrency, and EOT nodes for the pauses (i.e., state boundaries). Each node is annotated with an execution cost in processor clock cycles. In our case, these costs are derived using the technique presented in [69].

The intuitive semantics of PRET-C are illustrated using the execution traces in Table 2.1. We assume the preemption condition is false unless stated otherwise in the event column. Threads in a PRET-C program execute in a static order, from the first thread in the PAR statement to the last. For example, the PAR statement in the PRET-C code example (Fig. 2.1) spawns T1 and T2,
2.2 Timed Concurrent Control Flow Graph

Figure 2.2: A TCCFG running example. Used for demonstrating the formulations of WCRT algebra. The TCCFG is divided into boxes based on the hierarchy.
where $T1$ has higher priority than $T2$. This priority is encoded graphically in TCCFG in a similar fashion, where the left-most thread has the highest priority. The execution only switches to the next thread when the current one reaches an EOT node. The tick count advances when all the active threads have reached their respective EOT nodes.

<table>
<thead>
<tr>
<th>Tick count</th>
<th>Execution path</th>
<th>Events tick</th>
<th>Tick cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$B1 \rightarrow B2 \rightarrow B3 \rightarrow B4 \rightarrow B5$</td>
<td>Entering abort</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>$B3 \rightarrow B4 \rightarrow B6 \rightarrow B7 \rightarrow B9$</td>
<td>Forking $T1$ and $T2$</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>$B3 \rightarrow B4 \rightarrow B8 \rightarrow B10 \rightarrow B11$</td>
<td>$T1$ terminates</td>
<td>100</td>
</tr>
<tr>
<td>4a</td>
<td>$B3 \rightarrow B4 \rightarrow B12 \rightarrow B13 \rightarrow B14 \rightarrow B15$</td>
<td>$T2$ terminates, joining</td>
<td>89</td>
</tr>
<tr>
<td>4b</td>
<td>$B3 \rightarrow B14 \rightarrow B15$</td>
<td>Preemption</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>$B16 \rightarrow B17$</td>
<td>Program finishes</td>
<td>25</td>
</tr>
</tbody>
</table>

Execution begins from the start node $B1$ and subsequently reaches the abort-start node $B2$. An abort consists of two concurrent threads in a TCCFG: CheckAbort for checking the abort condition and the abort body AbortBody. For a strong abort, CheckAbort has a higher priority than AbortBody and vice versa for a weak abort. These two threads are both spawned by the abort start node. The preemption takes place when either of the threads reach the abort end node $B14$.

For the simulation shown in Table 2.1, in tick 1, CheckAbort and AbortBody pause at their EOT nodes $B4$ and $B5$ respectively. In tick 2, the execution resumes from these EOT nodes. The thread AbortBody spawns $T1$ and $T2$ using the fork node $B6$ and suspends itself. In tick 3, $T1$ terminates as it reaches the join node $B13$, and $T2$ pauses at the EOT node $B11$. In tick 4, we present two alternative scenarios. If preemption does not take place (tick 4a), $T2$ terminates and activates the join node $B13$, whereupon the EOT node $B15$ is reached. However, if preemption takes place (tick 4b), CheckAbort reaches the abort-end node $B14$, preempting all threads in the abort body, and thus pausing the program at $B15$. In either case, the program finishes in tick 5 by reaching the end node $B17$.

### 2.3 Implicit Path Enumeration Technique

IPET is a timing analysis technique developed for sequential programs. Since synchronous programs must be compiled into sequential C code as an intermediate step, IPET may be used for WCRT analysis. The input of IPET is a Sequential Control Flow Graph (SCFG). An example of SCFG is shown in Fig. 2.3. All the nodes and edges are labelled, with execution cost beside each node. For example, the condition node $B2$ has an execution cost of six processor cycles.

The key idea of IPET is to abstract the explicit execution order and model the SCFG using Integer Linear Programming (ILP). ILP is single objective optimisation technique. Each edge is represented by an integer variable in the ILP model, which denotes the number of times the control flow has passed through the edge. The ILP model itself consists of two parts, the ILP constraints and the objective function. Both parts are linear expressions constructed using the
2.3 Implicit Path Enumeration Technique

Figure 2.3: An example of SCFG for illustrating IPET.

integer variables. The ILP constraints model the control flow of the SCFG, i.e., the number of times the control flow enters a node must be equal to the number of times it exits the node. Entering a node is denoted by the number of times the control flow passes its inflow edges, and similarly, exiting a node is denoted by its outflow edges. Thus, for each node, the sum of its inflow edges (i.e., the corresponding variable in the ILP model) must equal to the sum of its outflow edges. This formulation generates one ILP constraint per node in the SCFG. For example, the following ILP constraints are created for the control flow of the SCFG in Fig. 2.3:

\[
\begin{align*}
B1: & \quad E1 = 1 \\
B2: & \quad E1 = E2 + E4 \\
B3: & \quad E2 = E3 \\
B4: & \quad E4 + E7 = E5 + E6 \\
B5: & \quad E6 = E7
\end{align*}
\]

These ILP constraints capture the relationship between edges based on the control flow of the SCFG. For example, when the control flow leaves the start node B1, it goes through E1 once, thus setting its value to ‘1’. Then, the control flow enters B2 and it passes on to either E2 or E4. Thus, E1 equals the sum of E2 or E4, where only one of E2 and E4 can be ‘1’ (i.e., receive the control flow). Note that edges are modelled as a set of integer variables in the ILP model, which abstracts away the explicit execution order between them. For example, given the ILP constraint \( E2 = E3 \), we cannot tell whether its \( E2 \rightarrow E3 \) or \( E3 \rightarrow E2 \). As a result of this abstraction (loss of information), one cannot reconstruct the SCFG from the ILP model, and
the ILP model can produce infeasible paths as the solution.

The other part of the ILP model is the objective function, which represents the calculation of program execution time. The execution cost is generated when the control flow enters a node, thus we can associate each edge with an execution cost: the cost of its target node. In timing analysis, the objective function is simply the sum of all the variables multiplied by their associate execution cost. For this SCFG, the objective function is as follows:

\[
Obj: 6 \times E_1 + 20 \times E_2 + 0 \times E_3 + \ldots + 3 \times E_8
\]

Finally, the ILP model is solved using an ILP solver. The goal of the ILP solver is to find integer values for all the variables in the ILP model such that the objective function (i.e., upper bound) is maximised while all the ILP constants are satisfied.

**Additional constraints**

The accuracy of the ILP model can be improved by adding more ILP constraints. For the SCFG in Fig. 2.3, given the ILP constraints for all the nodes and the objective function, the ILP solver will estimate the maximum possible execution time (i.e., the maximum value for the objective function) as infinite. This is due to the unbounded loop between \( B_4 \) and \( B_5 \), i.e., the values of \( E_6 \) and \( E_7 \) have no upper bound, resulting in \( E_6 = E_7 = \infty \), making \( Obj = \infty \). To bound the loop, we add an additional ILP constraint \( E_7 \leq 3 \) to the ILP model. Now, the control flow can pass through \( E_7 \) at most three times during execution. Hence, after adding the loop bound, the estimate for execution time is improved to being 30 processor cycles, generated with \( E_1 = 1, E_4 = 1, E_5 = 1, E_6 = 3 \) and \( E_7 = 3 \). In theory, adding more ILP constraints improves the modelling accuracy. However, this also results in a longer solving time in general.
Precision and scalability of Worst-Case Reaction Time (WCRT) analysis are like two weights on the different side of a balance. Tipping towards one side will inevitably worsen the other. This is a common belief for all timing analysis in general. Developing a WCRT analysis technique that is both precise and scalable is a challenging problem. In this chapter, we present our first attempt to solve this problem: a novel WCRT analysis technique called ILP\textsubscript{C}.

As discussed in Sec. 1.1.2, there are two approaches for WCRT analysis: Implicit path enumeration and explicit path enumeration. The difference between these two approaches, as their names suggest, is whether to model the execution paths as a set, or as sequences [45]. The order in sequences is important in WCRT analysis. It increases the precision when analysing concurrent threads, as it considers how these threads align with each other in each tick. This is known as tick alignment [70].

Unfortunately, modelling tick alignment through explicit path enumeration is a resource intensive task, as it quickly leads to state explosion. This is a well-known problem of this technique. As for implicit path enumeration, if we try to match the same modelling accuracy by encoding all the infeasible paths as exclusions, the solving time will increase exponentially [57, 61]. To tackle this problem, we raise the following question: is it necessary to model the tick alignment fully to compute a precise WCRT?

Through the case studies undertaken in this research, we have found that in a majority of cases only a small subset of infeasible paths needed to be added as exclusions, for implicit path enumeration to be as precise as explicit path enumeration. To demonstrate this, we used an Integer Linear Programming (ILP) model as the baseline, then we gradually refined it by excluding infeasible paths, but only added just enough exclusions such that the precision matched with
explicit path enumeration. This led us to develop an iterative analysis framework, called ILP\textsubscript{C}. Although this idea of iterative analysis was also used in [60], we were the first to propose the technique, and our technique ILP\textsubscript{C} is demonstrated as more scalable. The main contributions of this work are summarized below:

- We developed a new ILP formulation for TCCFGs, called ILP\textsubscript{base}. ILP\textsubscript{base} is specifically developed for synchronous programs that takes into account the specialties of the paradigm. It can be considered as a superset of the conventional ILP technique, Implicit Path Enumeration Technique (IPET) [49]. It not only models the basic control flow as in IPET, but also is able to directly encode the high level context, including fork, join, preemption and concurrent execution. For analysing a synchronous program, ILP\textsubscript{base} is more precise compared to IPET, yet easier to solve.

- We developed an alternative approach to resolve the tick alignment problem which avoids state explosion. Based on the observation that using explicit path enumeration can suffer from state explosion, we developed an ILP based model called \textit{tick expression}. Tick expressions allow us to analyse only a small relevant part of the program to verify the tick alignment problem, thus reducing the search space and preventing over-analysing.

- We developed an iterative framework, called ILP\textsubscript{C}, that combines ILP\textsubscript{base} and tick expressions to compute the WCRT. As the iteration progresses, the precision of the WCRT improves, which provides better usability to the user. On top of that, ILP\textsubscript{C} can match the precision of the state-of-the-art WCRT technique while requiring only a fraction of the analysis time. For instance, in our benchmarking, ILP\textsubscript{C} is the only technique that can complete the analysis of all the benchmark programs.

In the remainder of this chapter, we first present a motivating example in Sec. 3.1 to illustrate the tick alignment problem, and the basic principle of how we resolve it in ILP\textsubscript{C}. Then, we present a running example for explaining the formulation of ILP\textsubscript{C} in Sec. 3.2. Following that, we detail ILP\textsubscript{C} in Sec. 3.3. Then, we benchmark the proposed technique against state-of-the-art WCRT analysis techniques [57, 58, 59, 61] and present the results in Sec. 3.4. Finally, we conclude this chapter in Sec. 3.5.

### 3.1 Motivating example

Explicit path enumeration has already been used in the literature to solve the tick alignment problem. However, we find that it may have been over-analysing in general cases, thus resulting in state explosion. We propose that using a gradually refined ILP, we can achieve the same precision while avoiding state explosion. In this section, we present a motivating example to demonstrate the tick alignment problem with both the implicit and explicit path enumeration, and the principle of the proposed approach.
Let us consider a motivating example. Fig. 3.1 shows a Finite-State Machine (FSM) representation of a synchronous program which is comprised of the three concurrent threads \( T_1 \), \( T_2 \) and \( T_3 \). \( T_1 \) consists of three sequential states, and \( T_2 \) consists of two. \( T_3 \) starts from the state \( C_1 \) and can branch to either \( C_2 \) or \( C_3 \). In this example, we assume \( T_3 \) can take either branch arbitrarily. The execution cost, processor cycles, is annotated on the side of each state.

During execution, each thread takes exactly one transition in each tick, and the resultant execution trace is shown in Fig 3.2. The motivating example begins with \( T_1 \), \( T_2 \) and \( T_3 \) executing the states \( A_1 \), \( B_1 \) and \( C_1 \) respectively. Then in the second tick, all the threads take one transition, and the program advances to one of the two possible state combinations, depending on the branch in \( T_3 \). After that, the program continues in this tick-by-tick fashion.

Below the execution trace is the execution time of each tick. Since threads in a synchronous program are only logically concurrent (i.e., executing sequentially on the processor), the execution time of a tick is the sum of the execution time of all three threads. For example, the execution time of Tick 1 is the sum of \( A_1 \), \( B_1 \) and \( C_1 \), which is 25 cycles. For Tick 2, the execution time depends on the branch that \( T_3 \) takes.
In order to compute the WCRT, we need to find the longest possible tick. In the following subsections, we will demonstrate how the two existing approaches and the proposed approach compute this value.

### 3.1.1 Implicit path enumeration

Implicit path enumeration is simple and fast, which computes the WCRT without emulating the execution step by step. In this motivating example, the WCRT is computed by summing up the worst-case execution time of each thread [29, 49], which results in a WCRT estimate of 42 clock cycles, and the computation is as follows:

\[
WCRT = \max(T1) + \max(T2) + \max(T3) \\
= A3 + B1 + C2 \\
= 20 + 10 + 12 = 42
\]

Although the computation is fast, the resultant state combination \( A3, B1 \) and \( C2 \) is actually infeasible, as \( B1 \) and \( C2 \) never execute together, thus this computed WCRT never occurs during the actual execution.

### 3.1.2 Explicit path enumeration

Explicit path enumeration is a more complex approach [57, 58, 59, 61] that computes the WCRT by constructing the execution traces of the program (shown in Fig. 3.2). This approach can compute a more precise estimate, but at the expense of a much longer analysis time. If we apply explicit path enumeration to the motivating example, we can obtain a tighter WCRT estimate of 35 clock cycles. The computation of the analysis is as follows:

\[
\begin{align*}
A1 + B1 + C1 &= 25; \\
A2 + B2 + C2 &= 30; \\
A2 + B2 + C3 &= 28; \\
A3 + B1 + C1 &= 35; \\
A1 + B2 + C2 &= 25; \\
A1 + B2 + C3 &= 23; \\
A2 + B1 + C1 &= 30; \\
A3 + B2 + C2 &= 35; \\
A3 + B2 + C3 &= 33;
\end{align*}
\]

\[
WCRT = \max(25, 30, 28, 35, 25, 23, 30, 35, 33) = 35
\]

Constructing the complete execution traces requires exploring the full state space, which is subject to the state explosion problem.
3.1 Motivating example

3.1.3 The proposed approach

The key observation here is that the results from the two approaches, which are $A_3$, $B_1$ and $C_2$ for implicit path enumeration and $A_3$, $B_2$ and $C_2$ for explicit path enumeration, are only different in one: $T_2$. If we consider a slightly different scenario where the graph is identical but the execution cost of $B_2$ is 15 instead of 3, then both approaches will deliver the same output $A_3$, $B_2$ and $C_2$, but the implicit path enumeration will be much faster.

Inspired by this special scenario, we propose a new approach for WCRT analysis, which is an iterative narrowing of a sound over-approximation. This approach consists of two steps. In the first step, we quickly compute a WCRT estimate and its corresponding execution path using implicit path enumeration. Then, in the second step, this execution path is verified for tick alignment. If the ticks can be aligned, then the analysis finishes. Otherwise, we refine the WCRT calculation by excluding the detected infeasible state combination, and start over again. Applying this approach to the motivating example (Fig. 3.1), we can reach a precise estimate in five iterations as follows:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>First step</th>
<th>Second step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WCRT = $A_3 + B_1 + C_2 = 42$</td>
<td>Cannot align</td>
</tr>
<tr>
<td>2</td>
<td>WCRT = $A_3 + B_1 + C_3 = 40$</td>
<td>Cannot align</td>
</tr>
<tr>
<td>3</td>
<td>WCRT = $A_2 + B_1 + C_2 = 37$</td>
<td>Cannot align</td>
</tr>
<tr>
<td>4</td>
<td>WCRT = $A_2 + B_1 + C_3 = 35$</td>
<td>Cannot align</td>
</tr>
<tr>
<td>5</td>
<td>WCRT = $A_3 + B_1 + C_1 = 35$</td>
<td>Can align</td>
</tr>
</tbody>
</table>

The proposed approach is similar to explicit path enumeration, in the sense that it is essentially exploring all the possible state combinations, both the feasible and infeasible. However, the difference is that it explores these combinations in order, from longest execution time to shortest. By the definition of WCRT, the proposed approach can stop when it reaches the first feasible combination, which gives the proposed approach scalable performance. In this motivating example, the state combinations in the 4th and 5th iteration are equivalent in WCRT. The order of which is evaluated first is entirely random. The proposed approach will finish faster if the combination $A_3$, $B_1$ and $C_1$ is evaluated earlier in the 4th iteration. Moreover, the analysis time is further shortened if the cost of $B_2$ is 15 (the special case). Then we can reach a precise WCRT in just one iteration.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>First step</th>
<th>Second step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WCRT = $A_3 + B_2 + C_2 = 47$</td>
<td>Can align</td>
</tr>
</tbody>
</table>

As shown by this motivating example, using an iterative approach may significantly shorten the analysis time while also being precise. Based on this intuition, we are able to develop the new WCRT analysis technique ILP$_C$. 

Figure 3.3: A running example of Timed Concurrent Control Flow Graph (TCCFG).
3.2 TCCFG running example

Fig. 3.3 presents an example TCCFG, which will be used to demonstrate ILP$_C$. Each node is labelled with a unique ID prefixed with 'B', and each edge is labelled with a unique ID prefixed with 'E'. The number beside each node represents the execution cost of the node on a given processing platform. In this work, these costs are derived by using a well known approach presented in [69].

3.3 The proposed technique ILP$_C$

We demonstrated the idea of using iterative narrowing to achieve a precise and scalable WCRT analysis using a motivating example in Sec. 3.1. Based on this idea, we then developed a WCRT analysis technique, called ILP$_C$. In this section, we present the details of this technique.

The overview of ILP$_C$ is shown in Fig. 3.4. To fully utilise the high level context from the
synchronous program, we developed $ILP_C$ based on TCCFG instead of the conventional Sequential Control Flow Graph (SCFG). The analysis starts with the TCCFG of the program, and from it we create two models. The first model is $ILP_{base}$, which is an ILP model that captures the control flow of the program. This serves as the baseline for computing the WCRT, and the entry point of the iterative analysis. The second model is a collection of tick expressions, which are mathematical expressions derived from the structure of the TCCFG. These tick expressions capture the execution pattern of the program, and enable us to mathematically verify whether an execution path has a tick alignment problem. The final WCRT is computed by iteratively solving $ILP_{base}$ and tick expressions. In each iteration, if an execution path is verified to be infeasible, a new ILP constraint is generated and appended to $ILP_{base}$ to eliminate it, thereby improving the precision of the next iteration.

To deliver a scalable performance for $ILP_C$, we set two objectives in developing $ILP_{base}$ and tick expressions. The first is to capture the inherited high level context of the synchronous program, namely fork, join and preemption, in $ILP_{base}$ to enhance the precision, but only using minimal number of ILP constraints. The second is to capture the tick alignment in a divisible format (i.e., tick expressions), such that we can analyse only the relevant subset of information each time, thus preventing state explosion. In the following subsections, we will present the details of these two models.

### 3.3.1 The ILP model $ILP_{base}$

The ILP formulation for TCCFG is different from the conventional IPET. The conventional IPET is a formulation for the SCFG where an execution path is simply a connected path from the start node to the end node. For a TCCFG, the control flow is much more complicated. An execution path can be fragmented over multiple threads. For example, considering the running example in Fig. 3.3, when the fork node $B4$ executes, the control flow splits into two, denoting the concurrent execution of the spawned child threads, and at the join node $B15$, we must consider merging these two control flow back into one. In addition, we also have to consider preemption, which cannot only influences the control flow of the adjacent nodes, but all the threads in the abort body as well. Lastly, unlike in a SCFG which has only one start and end node that marks the beginning and end of an execution path, all EOT nodes in a TCCFG are marks for the beginning and end of an execution path. In $ILP_{base}$ we carry on the basic principle from the conventional ILP technique, and have created new formulations to accommodate these differences.

**The basic principle of $ILP_{base}$**

The underlying principle of $ILP_{base}$ is the same as in the conventional IPET [49]. The ILP model consists of an objective function which is the calculation of execution time, and a set of ILP constraints which models the control flow of the TCCFG. Both the objective function and ILP constraints are expressed in terms of variables, where each variable corresponds to an edge in the graph and represents the number of times the edge is triggered. The formulation
of the objective function is as follows:

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

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 associated the execution cost. Generally, in \text{ILP}_{\text{base}}, the execution cost of each node is associated with all its inflow edge(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, but it can execute just once, or may not execute at all depending on whether all the child threads have terminated. Hence, we add the execution cost of each join node to its outflow edge. For example, in Fig. 3.3 the cost of $B15$ is added to $E20$ instead of $E12$ and $E19$ (i.e., both $E12$ and $E19$ have 0 cost), and the associated cost of $E20$ is 15 (i.e., the sum of the cost of $B15$ and $B16$: 0+15).

Solving a ILP is to compute the values for all the variables such that the objective function is maximized, and the \textit{principle to model the control flow is to exclude what is infeasible using ILP constraints}. Initially, all the variables are independent of each other. We force the solver to produce a “correct” execution path by excluding patterns that are known to be infeasible. For example, if we include the ILP constraint ‘$E9 = E10$’, we have excluded all the solutions where $E8$ and $E9$ have different values, for we know that the control flow always enters and leaves the computation node $B8$ in the same tick. In the following subsections, we present our formulation for the ILP constraints.

\textbf{Assumptions and terminologies}

Our ILP formulation takes into account the following properties of synchronous programs and ILP solvers.

1. Synchronous programs do not have instantaneous loops and recursion [6]. Hence, each edge can be active at most once in each tick. Therefore, we bound the values of all the variables to be either ‘0’ (inactive) or ‘1’ (active) in \text{ILP}_{\text{base}} ($E_i \in \{0, 1\}$). We assume the instantaneous loops in external functions are substituted by equivalent computation nodes using techniques such as those shown in [71].

2. The objective function is a maximisation and all the edges have a positive contribution towards the objective function (i.e., all the execution costs are non-negative numbers). Thus, the ILP solver should always set the value of a variable to ‘1’ whenever possible. This monotonicity property is essential in our formulation, as it enables us to use inequality to model the control flow.
We also 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 Tn, let \( EOT(Tn) \) represents all the EOT edges of this thread. For example, in the example TCCFG (Fig. 3.3), \( EOT(T0) \) consists of E0 and E22, and \( EOT(T1) \) consists of E7 and E9.

### Computation and condition nodes

The control flow at computation and condition nodes is modelled in the same way as in the conventional technique: the inflow edges of a node are equal to the outflow edges. Let \( N_{conv} \) be an computation or condition node (i.e., conventional node) in the TCCFG, \( E_{in} \) and \( E_{out} \) be the inflow and outflow edges of \( N_{conv} \) respectively. The formulation for computation and condition nodes is as follows:

\[
\forall N_{conv} \in \text{TCCFG} \quad \sum E_{in} = \sum E_{out}
\]

The ILP constraint means that the number of times the control flow enters a node must be equal to the number of times it leaves the node. For example, the following ILP constraints should be generated for the nodes B9 and B9:

- B6: \( E7 = E8 \)
- B9: \( E10 = E11 + E12 \)

### EOT constraint

The inflow and outflow edges of EOT nodes have no direct correlation in a tick, since it takes one tick to transition from the inflow edges to the outflow edge. For EOT nodes, we generate ILP constraints to capture the mutual exclusion of their outflow edges (i.e., the EOT edges). Given the fragmental nature of an execution path in a TCCFG, it can start from multiple EOT nodes in a tick, and thus have multiple active EOT edges in the program. However, some combinations of EOT edges are infeasible. There are three types of infeasible combinations which are summarised as follows:

**EOT1** In each tick, each thread can have at most one active EOT edge (i.e., resuming execution from one of its EOT nodes). Hence, all the EOT edges in a thread are mutually exclusive.

**EOT2** The EOT edges in a parent thread and the EOT edges in the child threads are mutually exclusive. However, the EOT edges of a child thread are not exclusive from the EOT
3.3 The proposed technique ILP$_C$

edges of another child thread (i.e., sibling thread), since they can execute concurrently. Consider a scenario where a parent thread $P$ spawns $n$ child threads $T_n$. The EOT edges in each child thread $T_n$ is in conflict with the EOT edges in $P$.

EOT3 If a thread has more than one fork or abort-start nodes, only one of them can have active EOT edges.

To capture these rules, 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 grouped together. 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 analysing the hierarchy of threads in the TCCFG.

For example, Fig. 3.5 shows the thread hierarchy of the TCCFG in Fig. 3.3. The EOT edges of each thread can be grouped according to the thread hierarchy, from each leaf thread (e.g., $T_1$ and $T_2$) to the root thread (i.e., $T_0$). For example, the example TCCFG has the following three groups.

a. checkAbort ($E_3$) $\rightarrow$ $T_0$ ($E_0, E_{22}$)
b. $T_1$ ($E_7, E_9$) $\rightarrow$ abortBody (none) $\rightarrow$ $T_0$ ($E_0, E_{22}$)
c. $T_2$ ($E_{14}, E_{16}$) $\rightarrow$ abortBody (none) $\rightarrow$ $T_0$ ($E_0, E_{22}$)

The following ILP constraints are generated from the three groups to enforce EOT1 and EOT2.

\[ \text{a. } E_3 + E_0 + E_{22} \leq 1 \]
\[ \text{b. } E_7 + E_9 + E_0 + E_{22} \leq 1 \]
\[ \text{c. } E_{14} + E_{16} + E_0 + E_{22} \leq 1 \]
For this running example, we can only illustrate how to generate ILP constraints to enforce EOT1 and EOT2, as no thread has more than one fork/abort start node (i.e., no thread requires EOT3). If a TCCFG contains such a thread, additional ILP constraints can be generated to enforce EOT3 using a similar grouping technique to what we have presented here. 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.

**Fork constraints**

When a fork node executes, the control passes from the parent thread to all its child threads, which activates all the out/f_low edges of the fork node. Let \( N_{fork} \) be a fork node in the TCCFG, and \( E_{in} \) and \( E_{out} \) be the inflow and outflow edges of \( N_{fork} \) respectively. Then, the ILP constraints for modelling the control at the fork nodes are as follows:

\[
\forall N_{fork} \in \text{TCCFG}\\
E_{out} = \sum E_{in}
\]

For example, for the fork node \( B4 \) in Fig. 3.3, the following ILP constraints are generated.

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

**Join constraints**

The control flow at a join node is quite complex. The execution of a join node not only depends on its inflow edges, but also correlates to the state of the child threads inside. In our formulation, we associate the execution of a join node with its outflow edge, since the control flow only passes through when the join node executes, regardless of its inflow edge. This formulation generates two sets of ILP constraints, \( \text{Join1} \) and \( \text{Join2} \). In \( \text{Join1} \), we consider the join node with respect to the state of each of the child thread independently. Then in \( \text{Join2} \), we consider all the child threads as a whole.

<table>
<thead>
<tr>
<th>State of the child thread</th>
<th>Join node is allowed to execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alive and terminates in this tick</td>
<td>Yes</td>
</tr>
<tr>
<td>Alive but does not terminate in this tick</td>
<td>No</td>
</tr>
<tr>
<td>Not Alive</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.1: Truth table for join node with respect to one child thread

**Join1:** Let us first consider the execution of a join node with respect to only one of its child threads. A truth table that summarises this scenario is shown in Table 3.1. A child thread has three possible states in a tick. The first two are obvious: If the child thread is alive during
this tick, then the join node is allowed execution only if the child thread also terminates in this tick. However, if the child thread is not alive (i.e., the third row), we assume it to be terminated in the past and allow the join node to execute. The third row of the truth table is an overestimation for execution time, but it will be refined by $Join_2$.

Since the principle of ILP constraints is to exclude infeasible edge combinations, we only have to model the case which the join node cannot execute. That is, when a child thread is alive, but does not terminate (second row in Table 3.1). Let $(N_{fork}, N_{join})$ be a fork-join pair in the TCCFG; $E_{join\_out}$ be the outflow edge of $N_{join}$ (e.g., E20 for B15); $T_i$ be a nested child thread in $(N_{fork}, N_{join})$ (e.g., T1 for (B4, B15)); $E_{fork\_Ti}$ be the edge that spawns $T_i$ (e.g., E6 for $T_i$); and $E_{Ti\_join}$ be the edges that lead $T_i$ to $N_{join}$ (e.g., E12 for T1). The ILP formulation $Join_1$ is as follows:

$$\forall (N_{fork}, N_{join}) \in \text{TCCFG}$$
$$\forall T_i \in (N_{fork}, N_{join})$$
$$E_{join\_out} \leq 1 - \text{alive}(T_i) + \text{terminate}(T_i)$$

Given:

$$\text{alive}(T_i) = \sum EOT(T_i) + E_{fork\_Ti}$$
$$\text{terminate}(T_i) = \sum E_{Ti\_join}$$

<table>
<thead>
<tr>
<th>alive($T_i$)</th>
<th>terminate($T_i$)</th>
<th>$E_{join_out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not possible</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Truth table for $Join_1$

For each $N_{join}$, one ILP constraint is generated for each of its nested child threads $T_i$. The idea of the ILP constraints is simple. The value of $E_{join\_out}$ depends on whether the child thread $T_i$ is alive (i.e., $\text{alive}(T_i)$) and whether it terminates in the current tick (i.e., $\text{terminate}(T_i)$). Here, the values of $\text{alive}(T_i)$ and $\text{terminate}(T_i)$ can only be either ‘0’ or ‘1’, and the combination of $\text{alive}(T_i)$ being ‘0’ and $\text{terminate}(T_i)$ being ‘1’ is not possible by construction. The value of $E_{join\_out}$ is forced to be ‘0’ if $\text{alive}(T_i)$ is ‘1’ and $\text{terminate}(T_i)$ is ‘0’. Otherwise $E_{join\_out}$ is set to ‘1’ by the solver. The logic of the ILP constraint is summarised in the truth Table 3.2.

The function $\text{alive}(T_i)$ computes whether $T_i$ is alive in the current tick (i.e., has an active edge), which is formulated as the sum of all the EOT edges in the $T_i$ plus the edge that spawns $T_i$. We use these edges for two reasons. First, if there is an active edge in $T_i$, then at least one of these edges is ‘1’ since the execution of $T_i$ can only be initiated from either its EOT nodes (i.e., an active EOT edge), or from the fork node (i.e., active $E_{fork\_Ti}$). Second, using these edges ensures the output of the function is either ‘1’ or ‘0’. The EOT constraints ensure that all the EOT edges in $T_i$ are mutually exclusive; and $E_{fork\_Ti}$ (the parent thread has an active EOT edge) is exclusive from all the EOT edges, thus making the sum in $\text{alive}(T_i)$ to be either ‘1’ or ‘0’.

The function $\text{terminate}(T_i)$ computes whether $T_i$ terminates in the current tick, which is for-
mulated as the sum of $E_{Ti,\text{join}}$. The value of the sum is at most ‘1’, since a thread can terminate only once in each tick in a correct synchronous program. If the $T_i$ reaches the join node $N_{\text{join}}$, the output of $\text{terminate}(T_i)$ will be ‘1’, otherwise ‘0’.

As an example, the Join1 ILP constraints for the running example are as follows:

$$T1: E_{20} \leq 1 - (E_{7} + E_{9} + E_{6}) + (E_{12})$$
$$T2: E_{20} \leq 1 - (E_{14} + E_{16} + E_{13}) + (E_{19})$$

In this formulation, the outflow edge of the join node can only be active if each of its child threads allows it. The less than or equal sign is used to allow each child thread to restrict the value of $E_{\text{join\_out}}$ independently without interfering with each other.

**Join2**: In Join1, we consider each of the child threads independently, based on their state as shown in Table 3.1. There is a special scenario: if all child threads are not alive, the join node should not execute. For a join node to execute, at least one thread must be alive and reach it in the current tick. We use the formulation of Join2 to exclude this scenario. Let $N_{\text{join}}$ be a join node in the TCCFG, and $E_{\text{in}}$ and $E_{\text{out}}$ be the inflow and outflow edges of $N_{\text{join}}$ respectively. Then, the formulation Join2 is as follows:

$$\forall N_{\text{join}} \in \text{TCCFG}$$
$$E_{\text{out}} \leq \sum E_{\text{in}}$$

This formulation ensures that if no child thread reaches the join node in this tick, the join node does not execute. The ILP constraints force the outflow of the join node $E_{\text{out}}$ to be ‘0’ if none of the inflow edges are active, otherwise $E_{\text{out}}$ is allowed to be ‘1’. The less than or equal sign is used here to allow coexisting with Join1. Also, because the sum of the inflow edges of the join node can be greater than ‘1’.

For the running example, the ILP constraint Join2 is as follows:

$$B15: E_{20} \leq E_{12} + E_{19}$$

**Unreachable Join**: If a join node cannot be active by construction, for example, one of the child threads can never reach the join node and thus the join node never executes, then the nodes after the join node which cannot be reached by other means are removed by the Precision Timed C (PRET-C) tool chain, and the join node has no outflow edge. In this case, no ILP constraint is needed.
3.3 The proposed technique ILP C

Preemption

In a TCCFG, preemption is captured using a pair of abort-start and abort-end nodes. For example, \( B1 \) and \( B16 \) in the running example (Fig. 3.3) are a pair of abort-start and abort-end nodes. These two nodes always have two child threads, and have a structure as shown in Fig. 3.6. These two threads can be identified based on their priority, namely the high priority thread \( T_{\text{high}} \), and the low priority thread \( T_{\text{low}} \). In a strong abort, the abort condition is checked in \( T_{\text{high}} \) (i.e., \( T_{\text{high}} \) is the \textit{checkAbort} thread), and for a weak abort, it is checked in \( T_{\text{low}} \). The preemption takes place when either of the threads reaches the abort-end node, and the control flow immediately continues from the abort-end node onward. However, which thread reaches the abort-end node makes a difference in the execution path. If \( T_{\text{high}} \) reaches the abort-end node, i.e., \( E_{\text{preemption}} \) is active, everything in \( T_{\text{low}} \) is preempted (i.e., no active edge in \( T_{\text{low}} \)). If \( T_{\text{low}} \) reaches the abort-end node, \( T_{\text{high}} \) can still have active edges as it always executes before \( T_{\text{low}} \). To force a “correct” execution path, we only need to model the first scenario where \( T_{\text{low}} \) is preempted.

In the following formulation, we will continue to use the notions \( T_{\text{high}} \), \( T_{\text{low}} \) and \( E_{\text{preemption}} \) to denote the corresponding elements in the TCCFG. To capture the preemption, three sets of ILP constraints are generated: (1) for modelling the control flow at abort-start node (2) abort-end nodes, and (3) for modelling \( E_{\text{preemption}} \) preempts \( T_{\text{low}} \).

\textbf{Abort-start:} Let \( N_{\text{abortS}} \) be an abort-start node in the TCCFG, \( E_{\text{in}} \) be the inflow edges of \( N_{\text{abortS}} \), \( E_{\text{high}} \) be the outflow edge of \( N_{\text{abortS}} \) that spawns \( T_{\text{high}} \), \( E_{\text{low}} \) be the outflow edge that spawns \( T_{\text{low}} \). The formulation for abort-start nodes is as follows:

\[
\forall N_{\text{abortS}} \in \text{TCCFG} \quad \begin{align*}
E_{\text{high}} &= \sum E_{\text{in}} \\
E_{\text{low}} &\leq \sum E_{\text{in}}
\end{align*}
\]
An abort-start node is like a fork node which spawns two threads. The values of its outflow edges depend on its inflow edges. The edge $E_{hi}$ is always active when the abort-start node is reached. However, the edge $E_{lo}$ is only active if $T_{lo}$ is not preempted. Thus, the less than or equal sign is used for $E_{lo}$, which allows the ILP solver to set $E_{lo}$ to ‘1’ (prioritised over ‘0’), but also allowing it to be ‘0’ without constraining $E_{in}$ and $E_{hi}$. The less than or equal sign in the formulation enables us to create additional ILP constraints for modelling the preemption, which sets $E_{lo}$ to ‘0’ if preemption takes place. For the abort-start node $B1$ in the running example, the following ILP constraints are generated:

$$B1: E1 = E0$$
$$E5 \leq E0$$

**Abort-end nodes:** The control flow at an abort-end node is similar to a computation node. An abort-end node executes when any of the child threads reaches it. In other words, its outflow edge is active (An abort-end node has only one outflow edge) if any of its inflow edges is active.

Let $N_{abortE}$ be an abort-end node in the TCCFG, $E_{in}$ be the inflow edges of $N_{abortE}$, and $E_{out}$ be the outflow edges. The formulation for abort-end nodes is as follows:

$$\forall N_{abortE} \in \text{TCCFG}$$
$$E_{out} = \sum E_{in}$$

The value of the left-hand side of the ILP constraint is bound to be either ‘0’ or ‘1’ since $E_{out}$ is a binary variable, which implies that at most only one of the inflow edges of an abort-end node can be active (i.e., the right-hand side can sum up to ‘1’ at most). This is correct since they cannot be active together due to preemption. As an example for the formulation, the following ILP constraint is generated for the abort-end node $B15$ in the running example:

$$B16: E21 = E4 + E20$$

**Preemption of $T_{lo}$**

When $E_{preemption}$ is active, $T_{lo}$ and all its nested child threads are preempted. An equivalent scenario in the running example is when $E4$ is active, the abortBody (i.e., $T_{lo}$) and all its nested child threads $T1$ and $T2$ must have no active edge. In other words, all the edges in abortBody and its nested child thread are in conflict with $E_{preemption}$. The conventional technique will be creating one ILP constraint for each of the conflicting pairs, however this is very inefficient [56]. We have developed an alternative formulation to achieve the same result with far less ILP constraints. Let $(N_{abortS}, N_{abortE})$ be an abort pair in the TCCFG, $T_i$ be a thread in $(N_{abortS}, N_{abortE})$ excluding $T_{hi}$, $E_{lo}$ be outflow edge of $N_{abortS}$ that spawn $T_{lo}$. The formula-
tion for preemption is as follows:

$$\forall (N_{\text{abortS}}, N_{\text{abortE}}) \in \text{TCCFG}$$

$$\forall T_i \in (N_{\text{abortS}}, N_{\text{abortE}}) - T_{\text{high}}$$

$$\text{EOT}(T_i) + E_{\text{low}} \leq 1 - E_{\text{preemption}}$$

The first step of the formulation is to find all the abort pairs. Here, $T_i$ is a thread in the set of all the preemptable threads of the abort pair (i.e., $T_{\text{low}}$ or one of its nested child thread). The left-hand side of the ILP constraint captures all the edges in a preemptable thread, and the right-hand side captures the preemption condition. Instead of including all the edges in the preemptable thread in the ILP constraint, which will introduce unwanted restriction between the edges, we use only EOT edges and the edge that spawns $T_{\text{low}}$. This reduces the complexity of the overall model, while achieving the same result.

As an example, the preemption in the running example is captured using the following ILP constraints:

**Abort pairs:** $(B1, B16)$

$$T1: (E7 + E9) + E5 \leq 1 - E4$$

$$T2: (E14 + E16) + E5 \leq 1 - E4$$

**abortBody:** $E5 \leq 1 - E4$

These ILP constraints are specially formulated to be in conjunction with other ILP constraints in $\text{ILP}_{\text{base}}$. For example, the ILP constraints for EOT nodes ensure that the left-hand side of each constraint is either ‘0’ or can sum up to ‘1’ at most, which matches the value range of the right-hand side. If $E4$ is ‘0’, then the threads $T1$ and $T2$ are constrained by the other ILP constraints in $\text{ILP}_{\text{base}}$. Otherwise if the $E4$ is ‘1’, the preemption takes place, then all the edges on the left-hand side are ‘0’ (i.e., all the edges in the preemptable threads are ‘0’).

### 3.3.2 Tick expressions

Synchronous programs are predicable because they exhibit repeatable execution. We attempt to solve the tick alignment problem by exploiting this repeatable pattern. The result is a mathematical model called *tick expression*. Tick expressions are fundamentally different from the conventional explicit path enumeration. It captures the pattern of execution, whilst explicit path enumeration is trying to derive the end results of the pattern.

**Execution instances:**

Ticks are marked by EOT nodes. To determine the tick alignment, we need to formalise the execution pattern of EOT nodes. The idea is to choose a reference node (i.e., tick 1), and describe the execution instances of subsequent nodes as *number of ticks elapsed* since reaching the ref-
ereference node. Since the tick alignment problem only exists between concurrent threads, we use the top-level fork/abort-start node as our reference node. This gives the greatest coverage for our formulation. For example, the reference node for the running example (Fig. 3.3) is the abort-start node B1. Its execution marks tick 1. Then the execution instance of the subsequent nodes can be described as follows:

\[
\begin{align*}
B1: & 1 \\
B4: & 1 \\
B5: & 1, 3, 5, \ldots \\
B7: & 2, 4, 6, \ldots \\
B10: & 1, 3, 5, \ldots \\
B12: & 2, 4, 6, \ldots \\
& \ldots
\end{align*}
\]

The fork node B4 executes in the same tick as the reference node B1, thus its execution pattern is 1 (i.e., executing in tick 1). Similarly, the first execution instance of the EOT nodes B5 and B7 are tick 1 and tick 2 respectively. Since B5 and B7 are connected as a loop, their execution repeats indefinitely in a step size of two. As shown by this example, the execution of a node can be described as a number series. By comparing these number series, we can verify for tick alignment. For example, for the concurrent threads T1 and T2, B5 can align with B10 as the intersection of their number series is not empty. However, B5 and B12 can never align.

**General form of tick expressions**

As the number series that describe the executions of nodes are monotonic, they can be captured as polynomial functions. A general form is as follows:

\[
C + \sum_i V_i \times X_i(M)
\]

\[
C, V_i, X_i, M \in \mathbb{Z}^+
\]

The terms C, V_i and M are constants, and X_i is a variable. C denotes the first instance that a given node can be reached, which is the first number of the number series. The term V_i denotes the repetition pattern of the node. Depending on the loops that the node is part of, a node can have multiple V_i to describe its complex behaviour. The term X_i is the variable of the function. For every possible value for X_i, the function computes a number in the number series. Lastly, the min property M denotes the earliest execution time of a node (i.e., lower bound of the number series). This term is particularly important for join nodes, as they do not necessarily execute when the control flow reaches them (i.e., denoted by C). All of the terms are non-negative integers \(\mathbb{Z}^+\).

As an example, let’s consider the tick expression “2 + 3X(10)”, which has a constant term 2, a variable term 3X, and its min property is 10. For \(X = \{0, 1, 2, 3, 4, 5, \ldots\}\), the function produces the number series “11, 14, 17…” (numbers must be greater than M).
Fig. 3.7 shows the running example with tick expressions. The computation and condition nodes are abstracted in this TCCFG to provide a clearer view of the execution. The tick expressions are displayed inside the nodes.

### 3.3.3 Verifying an execution path for tick alignment

Tick expressions have two distinct features. First, their general form can be reassembled as ILP constraints, which allows us to formulate the tick alignment problem using ILP. Second, they enable us to reason the execution of a node without needing any context of its preceding or subsequent nodes. By taking advantage of these two features, we can develop an effective technique for verifying the tick alignment problem.

An overview of our verification technique is shown in Fig. 3.8. Our technique resolves around
extracting the tick expressions of the participating nodes in an execution path, and using these to construct a simplified model of the overall program behaviour, then use this model to check whether the execution path is feasible. The verification process starts with an execution path computed by ILP_{base}. Then the participating EOT and join nodes are identified, and their tick expressions are extracted. Using the extracted tick expressions, a small ILP model called ILP_{check} is generated. The tick expressions capture the instants of time in which a node can execute. ILP_{check} is used to check if the execution of a set of nodes may overlap in time. If ILP_{check} has an integer solution, then we conclude the ticks can align, and the execution path is feasible, otherwise it is infeasible.

Formulation of ILP_{check}

The execution of a node is a number series, which can be considered as a function \( f(X_1, X_2, \ldots) \). The principle of ILP_{check} is to find a positive integer \( t \) which all the functions of the participating EOT and join node can produce. Let \( N \) be an EOT or join node in an execution path, \( f_N \) be the number series of \( N \). Then the formulation of ILP_{check} is as follows:

\[
\forall N \in \text{execution path} \\
t = f_N(X_1, \ldots)
\]

Solving the ILP model is equivalent to verifying for \( \exists t \in \mathbb{Z}^+ : X_i \in \mathbb{Z}^+ \). The objective function of ILP_{check} is not important as we are not finding an optimal solution. We set it to be the sum of all the variables with coefficients of ‘1’, and the goal is to maximise it. For solving ILP_{check}, we enable the break at first setting of the ILP solver, which stops the analysis when the first integer solution is found.
The next step is to derive the function \( f_N(X_1, \ldots) \) from the tick expressions. For simplicity, we consider a tick expression as one set with two integers \( \{c, \{v_1, v_2 \ldots v_m\}, \text{min}\} \), where \( c \) represents the constant term, \( \{v_1, v_2 \ldots v_m\} \) represents the variable terms, and \( \text{min} \) represents the min property. For a node which has the only one tick expression, the translation is straightforward:

\[
\begin{align*}
  t &= cV_c + v_1V_{v_1} \ldots v_nV_{v_n} \\
  V_c &= 1 \\
  t &\geq \text{min} \\
  t, V_c, V_{v_i} &\in \mathbb{Z}^+
\end{align*}
\]

For example, the tick expression “\( 3 + 3X + 4X(4) \)” can be translated into the following ILP constraints:

\[
\begin{align*}
  t &= 3V_c + 3V_{v_1} + 4V_{v_2} \\
  V_c &= 1 \\
  t &\geq 4 \\
  t, V_c, V_{v_1}, V_{v_2} &\in \mathbb{Z}^+
\end{align*}
\]

Given \( V_{v_1} \) and \( V_{v_2} \) are free to be any positive integers and ‘4’ is the lower bound, the value of \( t \) can be “6, 7, 9, 10, 11…” This is the number series the tick expression is representing. Thus the ILP formulation reassembles \( t = f(X_1 \ldots) \).

For a node that has multiple tick expressions, the formulation is more complex. The node can execute if \( t \) can satisfy any of the tick expressions. To model this in ILP, we need to use a special function called a \( \text{SOS} \). In a \( \text{SOS} \), only one variable is allowed to have a non-zero value. For example, \( \text{SOS}(V_1, V_2 \ldots, V_n) \) constrains that only one of the variables in \( \{V_1, V_2 \ldots V_n\} \) can have non-zero positive integer value.

For a node \( N \) which has multiple tick expressions, each tick expression is actually a function \( f_{N_i}(V_1, V_2 \ldots) \). Then the idea of modelling \( f_N(V_i \ldots) \) in ILP is as follows:

\[
\begin{align*}
  f_N &= f_{N_1}(V_{1i}, \ldots) + f_{N_2}(V_{2i}, \ldots) + \ldots + f_{N_n}(V_{ni}, \ldots) \\
  \text{SOS}(f_{N_1}, f_{N_2} \ldots f_{N_n})
\end{align*}
\]

This will allow the ILP solver to freely choose any of the tick expressions.
Based on the idea of SOS, the general ILP formulation for a node $N$ with multiple tick expressions can be refined as follows:

Given:

$te_i \in$ Tick expressions of $N$

and $te_i = c_i V_{ci} + v_{i1} V_{v11} + v_{i2} V_{v12} \ldots v_{im} V_{vim}$

$$t = \sum_i te_i$$

$SOS(te_1, te_2 \ldots)$

$$\sum_i V_{ci} = 1$$

$$t \geq \sum_i min_i V_{ci}$$

$t, te_i, V_{ci}, V_{vij} \in \mathbb{Z}^*$

When we consider tick expressions as a group, exactly one constant term in the group is ‘1’. To capture this, we sum the ILP variables that correspond to the constant terms and make the sum equal to ‘1’ (i.e., $\sum_i V_{ci} = 1$). Combining with the SOS, we force the solver to pick exactly one tick expression from the group, i.e., exactly one $te_i$ will be non-zero. Moreover, the variable for a constant term becomes a flag indicating which tick expression is in effect. For example, if $V_{c5}$ is ‘1’, then $te_5$ will be non-zero, and all other $te_i$ must be zero, and consequently all the $V_{ci}$ in them also must be zero. Based on this knowledge, we mimic the SOS functionality for the min property. In $t \geq \sum_i min_i V_{ci}$, only one $min_i V_{ci}$ will be non-zero, and the others must be zero. This creates a dynamic lower bound for $t$ which is synchronised with the non-zero $te_i$. This synchronization is not possible if we put all $min_i$ into a SOS.

As an example for this formulation, consider a node has three tick expressions “$2 + 5X(2)$”, “$6 + 3X + 5X(10)$” and “$11 + 10X(11)$”. Then the following ILP constraints are generated:

$$te_1 = 2V_{c1} + 5V_{v11}$$

$$te_2 = 6V_{c2} + 3V_{v21} + 5V_{v22}$$

$$te_3 = 11V_{c3} + 10V_{v31}$$

$$t = te_1 + te_1 + te_1$$

$SOS(te_1, te_2, te_3)$

$$V_{c1} + V_{c2} + V_{c3} = 1$$

$$t \geq 2V_{c1} + 10V_{c2} + 11V_{c3} \quad \text{(Min property)}$$

$t, te_i, V_{ci}, V_{vij} \in \mathbb{Z}^*$
We have presented the formulation for ILP_{check} part by part. Let us consider a full example using our running example in Fig. 3.3. For instance, let us consider the following execution path of the running example:

\[ E3 \rightarrow E2 \rightarrow E7 \rightarrow E8 \rightarrow E16 \rightarrow E17 \rightarrow E18 \]

In this execution path, the participating EOT and join nodes are B3, B5 and B12. The tick expressions of these nodes are extracted, and the following ILP_{check} is generated:

\[
B3: t = 1 \times V_1 + 1 \times V_2 \quad \text{(i.e., } 1 + 1X (2)) \\
\quad t \geq 1 \\
B5: t = 1 \times V_3 + 2 \times V_4 \quad \text{(i.e., } 1 + 2X (1)) \\
\quad t \geq 1 \\
B12: t = 2 \times V_5 + 2 \times V_6 \quad \text{(i.e., } 2 + 2X (2)) \\
\quad t \geq 2 \\
\quad V_1 = 1, V_3 = 1, V_5 = 1 \quad \text{(constant terms)} \\
\quad t, V_1 - V_7 \in \mathbb{Z}^+ 
\]

The above ILP_{check} has no solution, as B5 never executes with B23. Hence, we conclude that the combination of nodes B3, B5 and B12 is infeasible during execution.

### 3.3.4 Eliminating the infeasible EOT combination

When an infeasible execution path is detected (i.e., the verification fails), a new ILP constraint is generated and added to ILP_{base} to eliminate the infeasible combination of nodes. Let \( E_i \) denote one of the \( n \) EOT edges in the execution path, then the following ILP constraint is generated:

\[ \sum_i E_i \leq n - 1 \]

The new ILP constraint sums up all the EOT edges in the execution path, and makes the sum less than the total number of them (less than or equal to the number of EOT edges minus one). This will force the ILP solver to remove or replace at least one EOT node. Note that join nodes are used in the verification process, but we do not include them in this additional ILP constraint.

Consider the execution path example presented in the previous section that has the EOT nodes B3, B5 and B12. The new ILP constraint to eliminate this infeasible path is as follows:

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

ILP\textsubscript{C} is radically different from all existing WCRT analysis techniques, and in principle, it is more scalable while being as precise as the existing techniques. To evaluate this hypothesis, we have benchmarked ILP\textsubscript{C} along with four other published WCRT analysis techniques. We present the results of these tests in this section.

The benchmarking is conducted in two phases. In the first phase, we stress all the techniques with synthetic TCCFGs to access their theoretical limits. The goal is to compare their scalability with respect to the increase in program size. Then, in the second phase, we compare the techniques using benchmark programs taken from [72, 73] to evaluate their performance in a more realistic setting.

3.4.1 The benchmarking settings

The benchmarking is carried out on a Windows-based machine using an Intel i7 4650U processor and 8 GB of RAM. The benchmark programs are in the form of TCCFGs (stored in XML format). For the first phase, these TCCFGs are directly generated by us for complete control over the stress test. In the second phase, we compile the programs for the MicroBlaze platform [74], and use the existing tool chain to construct TCCFGs from the binary [70].

The primary techniques we have evaluated are ILP\textsubscript{C}, an ILP-based technique ILP\textsubscript{S1} [57] and its subsequent work ILP\textsubscript{S2} [61], a model checking-based technique [58] and a reachability-based technique [59]. In the second phase, we also benchmarked IPET [49] as the baseline for comparing analysis time and precision. As IPET is known to be more generic but less precise, it is likely to produce a larger overestimate than the other techniques, thus it is not included in the first phase.

All the ILP based techniques (i.e., ILP\textsubscript{S1}, ILP\textsubscript{S2} and IPET) are formulated for SCFG. For these techniques, we convert the TCCFGs into SCFGs for using a compiler-like translation [75]. There are a few reasons for obtaining the SCFGs this way instead of directly constructing them from the binary. First, there is no actual binary in the first phase, thus converting from the TCCFGs is necessary. Second this approach gives us better control over the benchmarking process. Particularly, it guarantees the consistency in WCRT and control flow information (e.g., preemption, fork and join). This conversion is a behind-the-scenes process, so the time taken is not included as part of the analysis time for these techniques. Lastly, ILP\textsubscript{S1} and ILP\textsubscript{S2} involve generating an automaton during the analysis, so we made minor adjustments to adapt this step for the PRET-C semantics and matched their precision with other techniques.

As ILP\textsubscript{C} and the model checking [58] and reachability-based [59] techniques are natively developed for use with TCCFGs, we directly feed them our benchmarks.
3.4 Results

3.4.2 Synthetic benchmarking for scalability

In this first phase, we evaluate the scalability of all the techniques using systematically generated synthetic TCCFGs. In general, the number of program states increases exponentially as the number of concurrent threads increases. The scalability of a technique refers to its ability to cope with this exponential growth.

However, ILP\textsubscript{C} is developed explicitly to bypass this exponential growth. As shown by the motivating example in Sec. 3.1, for the same program, the analysis time of ILP\textsubscript{C} can vary for different cost assignments, since the number of iterations required is different. Thus we specially designed these synthetic TCCFGs to elicit this property, which allows us to alter iterations of ILP\textsubscript{C}.

Fig. 3.9 shows the template of the synthetic benchmarks, which has two threads $T1_a$ and $T1_b$. The two sets of synthetic benchmarks are generated from this template. The first set of the
synthetic benchmarks is generated by replicating these two threads alternatively. This symmetrical structure and execution cost assignment creates the worst-case for ILP\(_C\) that maximises the number of iterations. The second set of synthetic benchmarks are generated by replicating the two threads, but the execution costs of all the computation nodes are set to '10' and the costs of all the condition nodes are set to '5'. This second set of benchmarks elicits the best-case of ILP\(_C\), where the WCRT can be found in the first iteration.

All five techniques are first benchmarked against the first set of the synthetic benchmarks, with an increasing number of threads. The results are shown in Fig. 3.10. The computed WCRTs from all the techniques are identical. ILP\(_C\) is at its worst-case performance with this set of benchmarks, and its analysis time is labelled with ILP\(_C\)\(_{\text{max}}\) in Fig. 3.10. The analysis time of all techniques increases exponentially as the number of threads increases, and eventually the analysis time becomes impractical to measure.

ILP\(_S2\) has the worst performance overall, and can handle the benchmarks for up to five threads. This poor scalability is a result of the huge internal ILP model. ILP\(_S1\) and ILP\(_C\) have similar performance, and can handle the benchmarks for up to 10 threads. ILP\(_S1\) is an alternative approach to ILP\(_S2\), which have smaller ILP models but have to solve many of them to compute the WCRT. The trend of ILP\(_C\) is not as smooth as the other techniques. This is because a different ILP\(_\text{check}\) is used in each iteration for verifying tick alignment, whose solving time varies independent of the exact combination of tick expressions.

Finally, the model checking and reachability-based techniques have the most scalable performance, and can handle the benchmarks for up to 16 and 17 threads respectively. This outstanding performance is expected as the underlying tools are well-designed for exploring states, which are capable of applying abstraction and optimisation on the fly.
When benchmarking the five techniques against the second set of synthetic benchmarks, ILP$_{S1}$ and ILP$_{S2}$, the model checking and the reachability-based techniques do not have an observable change in their analysis time. However, the analysis time of ILP$_C$, which is labelled with ILP$_{C\_min}$ in Fig. 3.10, is reduced to less than 0.01 seconds regardless of the increase in the number of threads.

These results confirm our hypothesis for the iterative approach. In the best-case, ILP$_C$ matches the precision of the other techniques while completing the analysis almost instantly. However, ILP$_C$ can also be far worse than the model checking and reachability-based techniques in the worst-case. An interesting observation here is that the analysis time of ILP$_C$ is more sensitive to the program structure and the execution cost distribution, rather than purely correlated to the number of program states like the other techniques.

However, such symmetrical structure and execution cost distribution of the synthetic TCCFGs can rarely be found in real-life programs. We therefore hypothesise that ILP$_C$ will perform close to the best-case scenario in a more realistic setting, which is carried out in the second phase. ILP$_{S1}$ shows a more scalable performance compared to ILP$_{S2}$, thus we will only use ILP$_{S1}$ in the second phase.

### 3.4.3 Realistic benchmarks

In the second phase, we benchmark ILP$_{S1}$, the model checking and reachability-based techniques, and ILP$_C$ against a set of benchmark synchronous programs. For comparison of improvements in precision, we also benchmark IPET and ILP$_{base}$ (part of ILP$_C$) as the baseline references.

The benchmark programs are taken from [72] and [73], and two synthetic programs, are added to illustrate the significance of tick alignment. The details of these programs and benchmark results are shown in Table 3.3. The biggest program is WaterMonitor, which has 3204 lines of C code and 40 concurrent threads. Here, the model checking-based technique is abbreviated as MC.

The WCRTs obtained from the four precise WCRT analysis are consistent, which validates the correctness of ILP$_C$. ILP$_{S1}$ has the steepest increase in analysis time. For the largest three programs, it takes more than an hour to analyse. As for the model checking and the reachability-based techniques, the analysis time increases as the state-space grows. Eventually, the analysis could not complete for RailroadCrossing and WaterMonitor, due to the analysis time being too long (> 1hr) or insufficient memory (labelled as N/A in Table 3.3). If we consider each technique with respect to what it can handle, the average analysis time of ILP$_{S1}$ is 37.58 seconds, 7.66 seconds for the model checking-based technique, and 6.76 for the reachability-based technique.

In contrast, ILP$_C$ is able to compute the WCRT of all the programs, including the large ones that the other techniques cannot handle, very efficiently. The number of iterations ranges
<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Threads</th>
<th>Precise WCRT analysis</th>
<th>Baseline WCRT analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Analysis Time</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ILP\textsubscript{s1}</td>
<td>MC</td>
</tr>
<tr>
<td>ChannelProtocol</td>
<td>591</td>
<td>7</td>
<td>4.43</td>
<td>2.52</td>
</tr>
<tr>
<td>Flasher</td>
<td>816</td>
<td>7</td>
<td>10.87</td>
<td>2.65</td>
</tr>
<tr>
<td>RobotSonar</td>
<td>962</td>
<td>7</td>
<td>10.4</td>
<td>5.46</td>
</tr>
<tr>
<td>Synthetic1</td>
<td>1287</td>
<td>7</td>
<td>33.68</td>
<td>12.06</td>
</tr>
<tr>
<td>Synthetic2</td>
<td>1293</td>
<td>7</td>
<td>30.61</td>
<td>12.82</td>
</tr>
<tr>
<td>DrillStation</td>
<td>1094</td>
<td>15</td>
<td>135.5</td>
<td>5.1</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>2302</td>
<td>25</td>
<td>&gt; 1hr</td>
<td>N/A</td>
</tr>
<tr>
<td>RailroadCrossing</td>
<td>2713</td>
<td>30</td>
<td>&gt; 1hr</td>
<td>N/A</td>
</tr>
<tr>
<td>WaterMonitor</td>
<td>3204</td>
<td>40</td>
<td>&gt; 1hr</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.3: Analysis time summary
from 1 to 73, and the average analysis time is under 1.5 seconds, with a peak of 4.43 seconds for the Synthetic1 program. These results also suggest that the analysis time of ILP_C does not correlate with the program size. For example, the analysis time for WaterMonitor is shorter than DrillStation regardless of having more threads and longer C code.

As for the baseline techniques IPET and ILP_base, their analysis times are less than a few seconds and increase with program size at a very slow rate. However, the WCRTs they produce are less precise than other techniques.

3.4.4 Analysis time and precision trade-off

In this section, we compare the trade-off made in all the WCRT analysis techniques. We use IPET as the base line technique for precision and analysis time, and compare it with the other techniques with respect to the increase in analysis time and the percentage of WCRT reduction. This information is summarised in Table 3.4. For ILP_S1, the analysis time is 112 times longer than IPET on average, and the WCRTs are around 30% more precise. Model checking and reachability achieve the same amount of precision improvement, but with only around 10 times the analysis time of IPET. We also observe the state explosion problem here. As the program size increases, the analysis time increases exponentially, while the percentage of improvement in WCRT does not deviate much. For example, in our most extreme case, an increase of 8 threads (from Synthetic2 to DrillStation) causes the analysis time of ILP_S1 to increase from 13 times to 564 times longer than IPET.

In comparison, ILP_C and ILP_base are far more effective techniques, as they achieve better precision with no increase in analysis time compared to IPET. In most cases, the analysis times of ILP_C and ILP_base are even shorter than IPET. This is because of the well-formulated ILP constraints which actually reduce the solving complexity. The data also suggests that tackling the tick alignment problem is less cost effective compared with modelling the high-level control流程，as the analysis time of ILP_C is around 1.5 times that of ILP_base, while it only achieves 3% more improvement in WCRT precision on average, although it should be noted that this improvement is up to 10% in some cases (e.g., Synthetic1).

3.4.5 WCRT versus iteration in ILP_C

ILP_C has to perform multiple iterations to obtain the most precise WCRT in some of the benchmark programs (Fig. 3.3). In these cases, the WCRT expresses an interesting relationship with the number of iterations. To illustrate this, we take all the programs that require more than three iterations, and plot their WCRT against the accumulated analysis time in Fig. 3.11. The WCRTs produced in each iteration are marked on the line.

Fig. 3.11 shows that WCRT precision improves with the number of iterations. This trend is best shown by the DrillStation (Fig. 3.11(b)) and Synthetic1 (Fig. 3.11(c)) programs, with 73 and...
<table>
<thead>
<tr>
<th>Name</th>
<th>Conventional ILP</th>
<th>ILP&lt;sub&gt;S1&lt;/sub&gt;</th>
<th>Model checking</th>
<th>Reachability</th>
<th>ILP&lt;sub&gt;C&lt;/sub&gt;</th>
<th>ILP&lt;sub&gt;base&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>WCRT</td>
<td>Time</td>
<td>WCRT</td>
<td>Time</td>
<td>WCRT</td>
</tr>
<tr>
<td>ChannelProtocol</td>
<td>1</td>
<td>100%</td>
<td>29.5</td>
<td>-17.9%</td>
<td>16.8</td>
<td>-17.9%</td>
</tr>
<tr>
<td>Flasher</td>
<td>1</td>
<td>100%</td>
<td>40.3</td>
<td>-30%</td>
<td>9.8</td>
<td>-30%</td>
</tr>
<tr>
<td>RobotSonar</td>
<td>1</td>
<td>100%</td>
<td>12.4</td>
<td>-13.8%</td>
<td>6.5</td>
<td>-13.8%</td>
</tr>
<tr>
<td>Synthetic1</td>
<td>1</td>
<td>100%</td>
<td>16.6</td>
<td>-41.7%</td>
<td>5.9</td>
<td>-41.7%</td>
</tr>
<tr>
<td>Synthetic2</td>
<td>1</td>
<td>100%</td>
<td>13.5</td>
<td>-33%</td>
<td>5.6</td>
<td>-33%</td>
</tr>
<tr>
<td>DrillStation</td>
<td>1</td>
<td>100%</td>
<td>564.5</td>
<td>-39.4%</td>
<td>21.3</td>
<td>-39.4%</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>1</td>
<td>100%</td>
<td>&gt;1000</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RailroadCrossing</td>
<td>1</td>
<td>100%</td>
<td>&gt;1000</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>WaterMonitor</td>
<td>1</td>
<td>100%</td>
<td>&gt;1000</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.4: Precision trade-off
Figure 3.11: WCRT vs accumulated analysis time for the benchmark programs
37 data points respectively. This is an interesting property as it ties the WCRT precision to the elapsed analysis time. This means ILP_C can be interrupted at any time and is still able to produce a reasonably precise and safe WCRT estimate.

This property is a huge advantage in practice. First off, the user does not need to decide on the trade-off for precision, which is difficult since the increase in analysis time is not known (i.e., not knowing the cost for the investment in precision, and thus not being able to judge whether the trade is acceptable). Second, the user can stop ILP_C as soon as it produces a WCRT that is shorter than the deadline.

3.5 Conclusions

Synchronous languages are essential for designing safety critical systems, yet precise WCRT analysis of large synchronous programs has been a long-standing challenge. The common belief is that precision and scalability are a trade-off in timing analysis, and all existing techniques follow this pattern. Therefore, inevitably we have to sacrifice precision in practice for large programs in exchange for an acceptable analysis time. This is a difficult challenge, and in this chapter we proposed a new innovative approach to the problem.

To tackle the precision and scalability dilemma, we partitioned WCRT analysis into two parts — execution path analysis and tick alignment approximation, and we specifically developed a suitable analysis technique for each part. By combining these two analysis techniques, we created the iterative analysis ILP_C, which is radically different from all existing techniques. Through benchmarking, we observed that ILP_C is as precise as all the published techniques whilst being orders of magnitude faster for large programs.

The contribution not only lies in the analysis techniques that we have developed, but is also an illustration of the potential of iterative analysis. One strong feature of iterative analysis is to an WCRT estimate without needing to explore the full state space. This is one potential solution to battling the ever-growing complexity of modern systems. In ILP_C we clearly demonstrate this potential.

Moreover, ILP_C meets the demand for today’s rapid system development. Unlike all existing WCRT analysis techniques, the precision of ILP_C scales with analysis time, saving users from the hassle of picking a precision from the beginning. This can potentially lead to a new design flow, since ILP_C can provide almost instant feedback on the timing of the system.
In Chapter 3, we demonstrated that implicit path enumeration can match the precision of explicit path enumeration with respect to tick alignment, while being more scalable. However, explicit path enumeration still has a few valuable features that are very hard to achieve with implicit path enumeration, for example, variable tracking, which helps in determining the exact evaluation results at branches for given inputs. In this chapter, we present our second attempt at the Worst-Case Reaction Time (WCRT) analysis problem: developing a scalable WCRT analysis technique based on explicit path enumeration.

One of the most well-known issues with explicit path enumeration is the state explosion problem, which is caused by the composition of concurrent threads. Using concurrent threads is a natural way to express many algorithms, and synchronous programs often contain many of them. Each thread may not be very large or complex. However, when using explicit path enumeration, we have to consider all possible state combinations of the concurrent threads, resulting in an exponential growth in search space.

Our solution to this problem is to develop a new technique based on min-max-plus algebra, called WCRT algebra, which models the Control Flow Graph (CFG) from a timing perspective. This work is based on a recently developed timed semantics of synchronous programs [76], which was for the timing analysis of an intermediate-level representation of the SCCharts language [77], called Input-Output Boolean Tick Cost Automata (IO-BTCA). This theoretical work addresses the use of min-max-plus algebra for obtaining sound abstractions of IO-BTCA, ranging from modelling of exact signal-dependent behaviour to fully signal-abstract models. However, the work only treats flat concurrent compositions of IO-BTCA [76]. It cannot han-
dle hierarchy and preemption. Furthermore, it also lacks methodologies for transforming between the CFG and IO-BTCAs, which is necessary for an actual implementation.

In this chapter, we propose a novel and practical WCRT analysis technique which combines the benefits of the explicit path enumeration approach with those of the algebraic setting. Existing explicit path enumeration techniques compute the WCRT by first constructing a functionally equivalent automaton from the program, and then enumerating its states [58,59,61]. While this is intuitive, it suffers from the state explosion problem. In the proposed technique, we exploit the abstraction power of timing equivalence in min-max-plus algebra in order to reduce the effects of state space explosion significantly.

The contributions of this work are the following:

- We propose Tick Cost Automata (TCA) for WCRT analysis based on the idea of timing equivalence and formalise the behavioural semantics using min-max-plus algebra, which we term WCRT algebra. This provides a sound and more direct mathematical formulation of WCRT analysis.

- We develop a transformation of Timed Concurrent Control Flow Graph (TCCFG) into Tick-Cost Automaton (TCA), which allows the TCA to be generated in a fully structural manner along the hierarchy of the program.

- We implement a timing simulation based on the WCRT algebra and experimentally compare its performance with the state-of-the-art model checking-based [58] WCRT analysis and ILP_C. On our benchmarks, the results show that our implementation of WCRT algebra is as precise as the other techniques whilst being considerably faster.

The remainder of this chapter is organised as follows: We first introduce a motivating example in Sec. 4.1. Then we present a brief overview of the intermediate format used for analysis, called TCCFG, in Sec. 4.2. Following that, we show our formalisation of TCA and the proposed WCRT algebra in Sec. 4.3. The experimental results are reported in Sec. 4.5. Finally, we make concluding remarks in Sec. 4.6.

4.1 Motivation

The core of WCRT analysis is to model the program mathematically; in our case, using automata. For the same program, the automata can be different depending on the perspective of the modelling. Since WCRT is a timing property, we model a program from a timing perspective. The execution of a synchronous program can be considered as a sequence of ticks \{Tick_1, Tick_2, Tick_3, \ldots\}, and in each tick \(i\) the program has a set of possible execution times \(\{C_{Tick_1}, C_{Tick_2}, C_{Tick_3}, \ldots\}\) corresponding to different branches in the program. The maximum execution time of tick \(i\) can be defined as \(C_{Tick_i} = \max\{C_{Tick_1}, C_{Tick_2}, C_{Tick_3}, \ldots\}\), and the WCRT is the largest execution time of all the ticks WCRT = \(\max\{C_{Tick_1}, C_{Tick_2}, C_{Tick_3}, \ldots\}\). When programs are
executed concurrently, assuming an interleaved (multi-threaded) execution, the timing costs are added to give the cost of a tick for the composite program. The fact that the maximum over sums is not the same as the sum over the maxima creates the crucial precision vs efficiency trade-off that we propose to handle using TCAs. In this section, we will illustrate the idea of TCA using a motivating example and compare it with the existing techniques.

Let us consider a synchronous program with two concurrent threads $T_A \parallel T_B$ as shown in Fig. 4.1. For simplicity, the two threads are identical in structure. Each thread has four states, with the initial states $A1$ and $B1$ respectively. In each tick, threads take exactly one transition, then wait for each other. The WCRT of this program is the maximal possible timing cost in any reachable tick in the execution of $T_A \parallel T_B$. We write this timing value as $\text{WCRT}(T_A \parallel T_B)$.

For this example, existing techniques require 11 operations to compute the WCRT, and the number of operations increases exponentially with the number of threads. This is a well-known aspect of explicit path enumeration. If we extend the motivating example with $n$
Timing analysis of synchronous programs using WCRT Algebra

threads of identical structure, the number of required operations is $n^2 + 2n - 1$.

- The existing max-plus approach. The state explosion is a result of the concurrent composition of threads. This can be mitigated by abstracting the threads before composing them. One well-known technique is the max-plus approach [78], which abstracts the threads into a single value before composition, i.e., estimating $\text{WCRT}(T_A \parallel T_B)$ as $\text{WCRT}(T_A) + \text{WCRT}(T_B)$. The computation is:

$$\text{WCRT} = (A_1 \oplus A_2 \oplus A_3 \oplus A_4) \odot (B_1 \oplus B_2 \oplus B_3 \oplus B_4). \quad (4.2)$$

While this abstraction greatly reduces the number of operations ($4n - 1$ for $n$ threads), the modelling accuracy is also reduced. For example, if $\text{WCRT}(T_A) = A_1$ and $\text{WCRT}(T_B) = B_2$, then $\text{WCRT}(T_A \parallel T_B)$ is estimated as $A_1 \odot B_2$. As shown in Fig. 4.2, this combination of states is infeasible (i.e., the tick alignment problem [58]). Thus, it is an overestimate of $\text{WCRT}(T_A \parallel T_B)$.

- The proposed technique using TCAs. It is a good idea to abstract threads before composition, however, the problem is how to preserve sufficient tick alignment to determine the exact value of $\text{WCRT}(T_A \parallel T_B)$. We propose to achieve this by considering the execution of a thread from a timing perspective.

An automaton is WCRT equivalent to the original program if it produces exactly the same timing sequence $\{C_{\text{Tick}1}, C_{\text{Tick}2}, C_{\text{Tick}3}, \ldots\}$. Therefore, we can construct a minimal WCRT equivalent automaton of the program, and compute the WCRT from it. We call this automaton a Tick-Cost Automaton (TCA). A TCA representation can be smaller and simpler than the control flow of the program, since only one state is needed for each tick, therefore the computation of WCRT can be much quicker.

The TCA of a thread can be derived using simple algebra. Fig. 4.3 shows the TCAs of the
4.2 Running example

Figure 4.3: Converting $T_A$ and $T_B$ into TCAs, and composing them to compute the WCRT.

threads $T_A$ and $T_B$. Each TCA has three states, with costs shown inside. If we compose these two TCA using state-wise additions, we obtain the TCA of the program, which also has three states. The WCRT computation using TCA is as follows:

\[
\text{Tick 1: } A_1 \oplus B_1 \Rightarrow A_4 \oplus B_4 \Rightarrow A_4 \circ B_4
\]
\[
\text{Tick 2: } (A_2 \oplus A_3) \circ (B_2 \oplus B_3) \Rightarrow (A_4 \oplus B_4)
\]

WCRT = $(A_1 \circ B_1) \oplus ((A_2 \oplus A_3) \circ (B_2 \oplus B_3)) \oplus (A_4 \circ B_4)$.  \hspace{1cm} (4.3)

By analysing the program from a timing perspective, we observe that the timing repeats itself every three ticks. The proposed technique can compute the WCRT from the TCA using only seven operations. More importantly, compared to the existing techniques, we do not lose any precision. If we expand the parentheses of the second term in Computation (4.3) using the distribution law, we obtain the exact same value as Computation (4.1). Finally, if we extend the example with $n$ threads of identical structure, the number of operations is $4n - 1$, which is the same as Computation (4.2).

Note that even though a TCA is equivalent to the conventional automaton when computing the WCRT, it is still an abstracted representation of the original program. Unlike a convention automaton (e.g., Fig. 4.1 or Fig. 4.2), the original program logic cannot be reconstructed from a TCA (e.g., from the three-states TCA of $T_A \parallel T_B$).
Timing analysis of synchronous programs using WCRT Algebra

Figure 4.4: A TCCFG running example. Used for demonstrating the formulations of WCRT algebra. The TCCFG is divided into boxes based on the hierarchy.
4.2 Running example

WCRT algebra is formulated based on Precision Timed C (PRET-C) [8] and its intermediate format Timed Concurrent Control Flow Graph (TCCFG) [70]. Fig. 4.4 shows a running example for this chapter, which will be used to explain the formulations.

4.3 The WCRT Algebra

Our WCRT analysis of TCCFGs is based on explicit path enumeration using TCA whose timing is captured by formal power series in min-max-plus algebra. We first present the mathematical structure of min-max-plus algebra in Sec. 4.3.1, then use it to define TCA in Sec. 4.3.2 and subsequently in Sec. 4.3.3–4.3.6 compute the TCA by recursion on the structure of the TCCFG.

4.3.1 Mathematical structure

Min-max-plus algebra \((\mathbb{N}_\infty, \wedge, \oplus, \circ, 1, 0)\) is defined over the set \(\mathbb{N}_\infty = \mathbb{N} \cup \{-\infty\}\) of execution times, where \(-\infty\) is used to denote inactive transitions or unreachable states. The three operators in the algebra are minimum \(\wedge\), maximum \(\oplus\), and \(\circ\) is addition. All three operators are commutative and associative. The notations \(\circ\) and \(\oplus\) are chosen to highlight their multiplicative and additive nature. The constants of the algebra are \(1 = \mathbf{0}\) and \(0 = \mathbf{-\infty}\). The element \(0\) is absorbant for the operator \(\circ\) and neutral for \(\oplus\), e.g., \(X \circ 0 = 0\) and \(X \oplus 0 = X\). The element \(1\) is neutral for \(\circ\), e.g., \(X \circ 1 = X\). The element \(0\) is also absorbant for the operator \(\wedge\), e.g., \(X \wedge 0 = 0\). The \(\wedge\) operator can be used to abstract a number \(X \in \mathbb{N}_\infty\) into a boolean \(X \wedge 1 \in \{0, 1\}\), e.g., \(X \wedge 1 = 0\) when \(X = 0\) and \(X \wedge 1 = 1\) when \(X \geq 1\). Powers are \(X^n = X^{n-1} \circ X\) for \(n \in \mathbb{N} \setminus \{0\}\) and \(X^0 = 1\).

Formal power series

We capture the infinite sequence of ticks in a program execution as formal power series. A formal power series is a polynomial function with an infinite number of terms, which has a general form of \(f(X) = \bigoplus_{i \geq 0} a_i X^i\). For a succinct presentation, the \(\circ\) between the coefficient \(a_i\) and \(X^i\) is often omitted. Each coefficient represents the timing of a tick. We can extract the coefficients from a formal power series using the notation \(a_i = [X^i]f(X)\).

A TCCFG does not have instantaneous loops and there is no dynamic creation of threads. Thus, the number of statements executed in any tick, and hence the maximal possible timing cost \([X^i]f(X)\) in any tick \(i \geq 0\) is statically bounded by the TCCFG. Moreover, the number of reachable programs states is finite. Hence, the coefficient sequence of \(f(X)\) must be ultimately periodic (e.g., Fig. 4.3). Thus, we can write a formal power series for some \(0 \leq n \leq m\) as
Timing analysis of synchronous programs using WCRT Algebra

follows:

\[
 f(X) = a_0 \oplus a_1 X \oplus \ldots \oplus a_{n-1} X^{n-1} \oplus \bigoplus_{i \geq 0} \left( a_n X^{i(m-n+1)+n} \oplus \ldots \oplus a_m X^{i(m-n+1)+m} \right)
\]

\[
 = a_0 : a_1 : \ldots : a_{n-1} : (a_n : \ldots : a_m)^\omega,
\]

where the exponent \( \omega \) denotes infinite repetition and the colon \( : \) is used as a compact way of writing finite sums of \( X^i \)-weighted coefficients \( a_i \) as a list. The first \( n \) coefficients (i.e., \( a_0 \) to \( a_{n-1} \)) occur only once in the execution. Then the coefficients repeat from \( a_{n+1} \) to \( a_m \). For instance, the TCCFG \( T_A \) in Fig. 4.3 generates the coefficient sequence

\[
 \bigoplus_{i \geq 0} \left( A1 X^3 \oplus (A2 \oplus A3) X^{3i+1} \oplus A4 X^{3i+2} \right) = (A1 : (A2 \oplus A3) : A4)^\omega
\]

which is an instance of (4.4) with \( n = 0, m = 2 \) and coefficients \( a_0 = A1, a_1 = A2 \oplus A3 \) and \( a_2 = A4 \).

The WCRT is the largest coefficient of \( f(X) \), which can be computed by setting \( X \) to 1. This effectively removes all the variable terms \( X^n \) since \( 1^n = 1 \) and leaves the \( \oplus \) operator to compute the maximum coefficient:

\[
 f(1) = a_0 \oplus (a_1 \oplus 1) \oplus \ldots \oplus (a_n \oplus 1^n) \oplus ((a_{n+1} \oplus 1^{n+1}) \oplus \ldots \oplus (a_m \oplus 1^m))^\omega
\]

\[
 = a_0 \oplus a_1 \oplus \ldots \oplus a_n \oplus (a_{n+1} \oplus \ldots \oplus a_m)^\omega = \max(a_0, a_1, \ldots a_m).
\]

Instead of \( f(1) \) we will also write \( f|_{X=1} \) for the evaluation of the series \( f(X) \) at \( X = 1 \).

Operations on formal power series

We will construct the timing equivalent TCA from a TCCFG \( G \) hierarchically based on the structure of \( G \) by algebraic operations on formal power series. A useful class of operations are the coefficient-wise variants of the binary operators \( \land \) and \( \circ \), denoted \( \land \) and \( \circ \), respectively, and the operator \( \oplus \) which is naturally coefficient-wise:

\[
 f_1(X) \circ f_2(X) = \bigoplus_{i \geq 0} \left( [X^i] f_1(X) \circ [X^i] f_2(X) \right) X^i
\]

\[
 f_1(X) \land f_2(X) = \bigoplus_{i \geq 0} \left( [X^i] f_1(X) \land [X^i] f_2(X) \right) X^i
\]

\[
 f_1(X) \circ f_2(X) = \bigoplus_{i \geq 0} \left( [X^i] f_1(X) \land [X^i] f_2(X) \right) X^i.
\]

Scalar multiplication with a constant \( C \) is given by

\[
 C \circ (f(X)) = C \circ (a_0 \oplus a_1 X \oplus a_2 X^2 \oplus \ldots)
\]

\[
 = (C \circ a_0) \oplus (C \circ a_1) X \oplus (C \circ a_2) X^2 \oplus \ldots
\]

\[
 = \bigoplus_{i \geq 0} (C \circ a_i) X^i = C^\omega \circ f(X),
\]
4.3 The WCRT Algebra

where \( C^\omega = \bigoplus_i C X^i \) is the constant power series with an infinite repetition of coefficient \( C \).

### Shifting the coefficients

When computing the WCRT, the timing of the current tick may depend on the timing in the previous tick. For example, the outflow of an EOT node depends on whether the EOT node is reachable in the previous tick. We use the \( \text{pre}(f(X)) \) function to produce a one-tick-delay variant of a formal power series \( f(X) \). This allows us to access the timing in the previous tick mathematically. The computation of \( \text{pre}(f(X)) \) is as follows:

\[
\text{pre}(f(X)) = (0 \oplus (f(X) \odot X))\tilde{\lambda}1^\omega.
\]

In this function, the term 0 is added to the series as the \( X^0 \) coefficient, while the \( \tilde{\lambda} \) operator and \( 1^\omega \) series reduces the coefficients of the series to be 1 or 0. For example, \( \text{pre}(10 : 3 : (0 : 43)^\omega) = 0 : 1 : 1 : (0 : 1)^\omega \).

### 4.3.2 Tick Cost Automata (TCAs)

From a timing standpoint, the program execution is a sequence of ticks, and each tick has two possible outcomes. The program can either pause at an EOT node and resume in the next tick, or reach the end node and terminate. TCAs capture timing based on this perspective. A TCA is formed of a sequence of states, and at each state there are two transitions: one leads to the next state denoting the cost to reach the next tick, and one leads to the end state denoting the cost to exit. Since the sequence is ultimately periodic, it will eventually loop back to one of the intermediate states. Fig. 4.5 shows the general form of a TCA.

Mathematically, we define a TCA \( \tau \) as two formal power series: \( \tau = (\text{tick}, \text{exit}) \), where \( \text{tick} \) captures the cost for reaching the next state (i.e., pausing at an EOT node), and \( \text{exit} \) captures the cost for reaching the end node. Given Fig. 4.5, these are:

\[
\tau = \begin{cases} 
\text{tick} & = a_1 : a_2 : \ldots : a_{n-1} : (a_n : \ldots : a_m)^\omega \\
\text{exit} & = b_1 : b_2 : \ldots : b_{n-1} : (b_n : \ldots : b_m)^\omega.
\end{cases}
\]
4.3.3 Worst Case Reaction Time Analysis

TCAs are the unit of composition in WCRT algebra. Accordingly, the WCRT of a synchronous program is obtained by constructing the TCA representation of its TCCFG. We compute the WCRT from this TCA. Given a TCA $\tau = (\text{tick}, \text{exit})$, the WCRT is

$$\text{WCRT}(\tau) = \text{tick}|_{X=\bot} \oplus \text{exit}|_{X=\bot}. \quad (4.5)$$

The equation (4.5) takes the maximum of the time cost, over all ticks, to reach a pause (i.e., $\text{tick}|_{X=\bot}$) or to terminate (i.e., $\text{exit}|_{X=\bot}$). The computation of the TCA associated with a TCCFG follows the hierarchical structure of the TCCFG. A TCCFG is a single (main) thread. A thread is a sequential control flow of nodes and boxes. A box consists itself of concurrent threads, thus generating a multi-level hierarchy. For instance, the main thread of the TCCFG in Fig. 4.4 has nodes $B_1, B_2, B_{14} - B_{17}$ and the box $\text{Box}_{\text{abort}}$, which comprises the threads $\text{CheckA}$ and $\text{ABody}$. Thread $\text{ABody}$ has nodes $B_5, B_6, B_{13}$ and box $\text{Box}_{\text{fork}}$. We call $\text{Box}_{\text{abort}}$ an abort box and $\text{Box}_{\text{fork}}$ a fork box.

We define a translation function $TCA(\Phi)$ which traverses this structure by recursive descent and maps each thread or box $\Phi$ into a timing equivalent TCA. The common feature making this possible is that each thread or box has a single start and a unique end point relative to which the timing can be measured out by a TCA. If $G$ is a TCCFG and $T_{\text{main}}(G)$ is the main thread, the WCRT of $G$ is

$$\text{WCRT}(TCA(T_{\text{main}}(G))). \quad (4.6)$$

In the following subsections, we first define the transformation of a simple thread into a TCA. Then we describe the flattening of fork and abort boxes.

4.3.4 Modeling the TCA of a thread

A thread can be converted into a TCA only if it has no boxes. If not, the boxes are flattened into a sequence of nodes using the techniques described in Sec. 4.3.5 and 4.3.6 later.

When a thread $T$ belongs to a fork box or an abort box (as opposed to the main thread), it is incomplete in the sense that it is a fragment of the TCCFG with no start and end nodes to mark the beginning and the termination of its execution. To make these threads self-contained, we add virtual start and end nodes to them. They are defined as $N_{\text{start}}(T) = T_{\text{start}}$ and $N_{\text{end}}(T) = T_{\text{end}}$, respectively. Fig. 4.6 and 4.7 shows the boxes $\text{Box}_{\text{fork}}$ and $\text{Box}_{\text{abort}}$, with all the virtual nodes. In Fig. 4.7, $\text{Box}_{\text{fork}}$ is replaced by the nodes $B_{1f_{\text{fork}}} - B_{6f_{\text{fork}}}$ as a result of flattening. These nodes are WCRT equivalent to original the box.

Additionally, when a thread belongs to a fork box, we add a virtual EOT node after its virtual end nodes to mimic the functionality of the join node. A thread terminates when its control flow reaches the end node. However, the joining does not have to take place at the same tick
4.3 The WCRT Algebra

Figure 4.6: TCCFG section \( Box_{fork} \): concurrent threads \( T1 \) and \( T2 \) with virtual start, virtual end and EOT nodes.
Figure 4.7: TCCFG section Box\textsubscript{abort}: the check abort threads CheckA and abort body ABody with virtual start and virtual end nodes. The box Box\textsubscript{fork} is replaced with a WCRT equivalent graph of $TCA(\text{Box}_{\text{fork}})$ indicated by the dashed box.
4.3 The WCRT Algebra

when this happens since sibling threads may still be active. This virtual EOT node serves as a memory of the termination, which is required for formulating the TCA of the box. This is a special EOT node which we denote as $N_{vEOT}(T)$. Those virtual nodes have 0 cost, thus they do not contribute to the WCRT (see Fig. 4.6).

When threads belong to an abort box, virtual EOTs are not required since preemption takes place instantaneously. An abort box always has two threads, one for checking the abort condition, and one for the abort body. The execution priority of these two threads depends on the type of the abort: strong or weak. We have developed a formulation that can handle both. In our formulation, we refer to the higher priority thread priority as $T_{high}$ and the lower priority thread as $T_{low}$. For example, $Box_{abort} = \{T_{high}, T_{low}\}$, where $T_{high} = CheckA$ and $T_{low} = ABody$.

Start node

A node is an atomic unit in a TCCFG, and its execution can be captured as a TCA. The start node $N_{start}(T)$ of a thread $T$ (e.g., $N_{start}(T1) = T1_{start}$ in Fig. 4.6) has the following TCA:

\[
\begin{align*}
tick(N_{start}(T)) &= 0^\omega \\
exit(N_{start}(T)) &= 1 : 0^\omega
\end{align*}
\]

This is a generic formulation for a start node. Since the execution of a synchronous program cannot pause at the start node, its formal power series tick constantly produces 0s across all ticks. The start node is the initiator of the program execution. Thus, in the first tick, it produces an execution cost of 1 in exit (i.e., the execution cost of the program is 1 by the time the control flow leaves the start node). Then, the start node remains 0 for the rest of the execution, i.e., no control flow leaves the start node.

Transient nodes

Transient nodes are computation, condition and end nodes. During execution, transient nodes contribute towards the execution cost when the control flow reaches them, and then they immediately pass on the control flow without pausing (i.e., the control flow exits the transient node in the same tick). Let $precede(N)$ be the set of predecessors of a node $N$ and $C_N$ the associated cost of $N$. The TCA of a transient node $N$ then is:

\[
\begin{align*}
tick(N) &= 0^\omega \\
exit(N) &= C_N \odot \left( \bigoplus \{exit(N_p) \mid N_p \in precede(N)\} \right).
\end{align*}
\]

The control flow of the program cannot pause at transient nodes, thus the series tick$(N)$ always produces 0. The exit cost of a transient node is the maximum cost received from its preceding
nodes plus its own cost. As an example, the TCAs of the node $B3$ and $B8$ are as follows:

\[
\begin{align*}
    \text{tick}(B8) &= 0 \\
    \text{exit}(B8) &= 25 \odot \text{exit}(B7) \\
    \text{tick}(B3) &= 0 \\
    \text{exit}(B3) &= 25 \odot (\text{exit}(B4) \oplus \text{exit}(\text{CheckA}_{\text{start}}))
\end{align*}
\]

**EOT nodes**

The EOT nodes are the state boundaries of program execution, which take one tick to execute. When the control flow reaches an EOT node, it pauses, and exits in the next tick. Depending on the context, the formulations for EOT nodes can be slightly different. The generic TCA of an EOT node $N$ is defined as follow:

\[
\begin{align*}
    \text{tick}(N) &= C_N \odot \left( \bigoplus \{\text{exit}(N_p) \mid N_p \in \text{precede}(N)\} \right) \\
    \text{exit}(N) &= \text{pre}(\text{tick}(N)).
\end{align*}
\]

The series $\text{tick}(N)$ denotes the maximum execution cost to pause at an EOT node, which is equal to the maximum cost from the preceding nodes plus the cost $C_N$ of the EOT node itself. The series $\text{exit}(N)$ denotes the exiting of the control flow, which is essentially a one-tick-delayed $\text{tick}(N)$ with an execution cost of 1. If an EOT node is reachable in a tick, i.e., $[X^n]\text{tick}(N) \geq 1$, then its $\text{exit}$ in the next tick is $[X^{n+1}]\text{exit}(N) = 1$, thereby resetting the execution cost to 0 at the beginning of a tick. If an EOT node is not reachable in a tick, i.e., $[X^n]\text{tick}(N) = 0$, then the exit will not be produced in the next tick, i.e., $[X^{n+1}]\text{exit}(N) = 0$.

For example, the TCA of the EOT node $B4$ is as follows (Fig. 4.6):

\[
\begin{align*}
    \text{tick}(B7) &= 10 \odot \text{exit}(T_{1,\text{start}}) \\
    \text{exit}(B7) &= \text{pre}(\text{tick}(B7))
\end{align*}
\]

**Virtual EOT in fork boxes** When a thread terminates in a fork box, the virtual EOT node is activated by receiving the control flow from the end node, and it remembers the termination through a self-loop. In other words, the virtual EOT nodes are the terminated states of the threads. Let us define $N_{\text{EOT}}(T)$ to be the set of EOT nodes of a thread $T$ and $\text{sibling}(T)$ the sibling threads of $T$, that is the thread which share the same box. The TCA of the virtual EOT nodes $N$ of a thread $T$ in a fork box $B$ is defined thus:

\[
\begin{align*}
    \text{tick}(N) &= C_N \odot \left( \bigoplus \{\text{exit}(N_p) \mid N_p \in \text{precede}(N)\} \right) \\
    \text{exit}(N) &= \text{pre}(\text{tick}(N)) \odot \left( \bigoplus \{\text{pre}(\text{tick}(N_s)) \mid N_s \in N_{\text{EOT}}(T_s), T_s \in \text{sibling}(T)\} \right).
\end{align*}
\]

The formulation is different from a normal EOT node as we have added a condition in $\text{exit}(N)$
to constrain the self-looping behaviour. The self-loop is useful when a thread terminates, while its sibling threads are still in the process of reaching their end nodes and have to pause at an EOT in this tick. The terminated state is remembered for a join in the future. The self-looping should become unreachable (i.e., \( \emptyset \)) again if none of the sibling threads can pause at an EOT node in the previous tick (i.e., \( \bigoplus \text{pre}(\text{tick}(N_i)) = \emptyset \) in the formulation). The self-loop is active only if, in the previous tick, the virtual EOT node was reachable (i.e., \( [X^n]\text{pre}(\text{tick}(N)) = 1 \)) and at least one sibling thread could reach an EOT node (i.e., \( \bigoplus [X^n]\text{pre}(\text{tick}(N_i)) = 1 \)). This prevents generating the TCA state where all the concurrent threads are self-looping at the virtual EOT node and no thread is actually executing. For example, the TCA of the virtual EOT node \( T_{1\text{vEOT}} \) is as follows (Fig. 4.6):

\[
tick(T_{1\text{vEOT}}) = 0 \odot (\text{exit}(T_{1\text{end}}) \oplus \text{exit}(T_{1\text{vEOT}}))
\]
\[
\text{exit}(T_{1\text{vEOT}}) = \text{pre}(\text{tick}(T_{1\text{vEOT}})) \odot (\text{pre}(\text{tick}(B9)) \oplus \text{pre}(\text{tick}(B11)))
\]

**EOT in abort boxes** In the case of an *abort* box \( B = \{T_{\text{high}}, T_{\text{low}}\} \), EOT nodes are modeled differently depending on whether they belong to the \( T_{\text{high}} \) or \( T_{\text{low}} \) thread.

- For the EOT nodes \( N_i \in N_{\text{EOT}}(T_{\text{high}}) \) in \( T_{\text{high}} \in B \) we have:

\[
tick(N_i) = C_{N_i} \odot \left( \bigoplus \{\text{exit}(N_p) \mid N_p \in \text{precede}(N_i)\} \right)
\]
\[
\text{exit}(N_i) = \text{pre}(\text{tick}(N_i)) \odot \text{pre}(\text{tick}(T_{\text{low}})),
\]

where we added a condition in \( \text{exit} \) to monitor the status of \( T_{\text{low}} \). Thread \( T_{\text{high}} \) resumes its execution from an EOT node only if in the previous tick (1) the EOT node was reachable (i.e., \( \text{pre}(\text{tick}(N_i)) \)) and (2) \( T_{\text{low}} \) can reach an EOT node (i.e., \( \text{pre}(\text{tick}(T_{\text{low}})) \)). If \( T_{\text{low}} \) cannot reach an EOT node in the previous tick \( n \), then \( T_{\text{low}} \) must have reached the abort-end node, preemption \( T_{\text{high}} \). Thus, \( T_{\text{high}} \) must not continue its execution in tick \( n + 1 \). For example, the TCA for the EOT node \( B4 \) in *CheckA* (Fig. 4.7) is as follows:

\[
tick(B4) = 10 \odot \text{exit}(B3)
\]
\[
\text{exit}(B4) = \text{pre}(\text{tick}(B4)) \odot \text{pre}(\text{tick}(Abody))
\]

- For all EOT nodes \( N_i \in N_{\text{EOT}}(T_{\text{low}}) \) in \( T_{\text{low}} \in B \) we define

\[
tick(N_i) = C_{N_i} \odot \left( \bigoplus \{\text{exit}(N_p) \mid N_p \in \text{precede}(N_i)\} \right)
\]
\[
\text{exit}(N_i) = \text{pre}(\text{tick}(N_i)) \odot (\text{tick}(T_{\text{high}}) \lambda 1^\omega)
\]

and for the start node of \( T_{\text{low}} \):

\[
tick(N_{\text{start}}(T_{\text{low}})) = 0^\omega
\]
\[
\text{exit}(N_{\text{start}}(T_{\text{low}})) = (1 : 0^\omega) \odot (\text{tick}(T_{\text{high}}) \lambda 1^\omega).
\]

Similar to the formulations for \( T_{\text{high}} \), we have added a condition in the \( \text{exit} \) of the EOT and start nodes in \( T_{\text{low}} \). The difference is that, here, we check the status of \( T_{\text{high}} \) in the current tick. \( T_{\text{low}} \)
can resume/initiate its execution from an EOT/start node only if $T_{\text{high}}$ can reach an EOT node in the current tick (i.e., $[X^n](\text{tick}(T_{\text{high}})) = 1$). If $T_{\text{high}}$ cannot reach an EOT node, then it must reach the abort-end node, preempting $T_{\text{low}}$ immediately; Thus, $T_{\text{low}}$ cannot execute after $T_{\text{high}}$ in the same tick. As an example, the TCAs for the EOT and start nodes in $A\text{Body}$ (Fig. 4.7) as follows:

\[
\begin{align*}
tick(ABody_{\text{start}}) &= 0^n \\
exit(ABody_{\text{start}}) &= (1 : 0^n) \circ (\text{tick}(\text{CheckA})) \\
tick(B5) &= 10 \cap exit(ABody_{\text{start}}) \\
exit(B5) &= \text{pre}(\text{tick}(B5)) \circ (\text{tick}(\text{CheckA})) \\
tick(B1_{\text{fork}}) &= 20 \cap exit(B6) \\
exit(B1_{\text{fork}}) &= \text{pre}(\text{tick}(B1_{\text{fork}})) \circ (\text{tick}(\text{CheckA})) \\
\end{align*}
\]

\[\ldots\]

**TCA of threads**

Finally, the TCA for a thread $T$ is simple. The series $\text{tick}(T)$ is the maximum execution cost to reach and pause at an EOT node, and $\text{exit}(T)$ is the maximum execution cost to reach the end node and exit:

\[
\begin{align*}
tick(T) &= \bigoplus \{ \text{tick}(N_i) \mid N_i \in N_{EOT}(T) \} \\
exit(T) &= exit(N_{end}(T)).
\end{align*}
\]

For example, the TCA of $T1$ and $T2$ in $Box_{\text{fork}}$ is as follows:

\[
\begin{align*}
tick(T1) &= \text{tick}(B7) \\
exit(T1) &= exit(T1_{\text{end}}) \\
tick(T2) &= \text{tick}(B9) \oplus \text{tick}(B11) \\
exit(T2) &= exit(T2_{\text{end}})
\end{align*}
\]

### 4.3.5 Modeling the TCA of a fork box

When the threads making a fork box $B$ are all reduced as TCAs, we can then use them to express the TCA of the fork box itself. This captures forking, concurrent execution and the
joining of threads:

\[
tick(B) = \bigcirc \{ tick(T_i) \oplus tick(N_{\text{EOT}}(T_i)) \mid T_i \in B \} \]  

\[
exit(B) = \bigcirc \{ tick(N_{\text{EOT}}(T_i)) \mid T_i \in B \} \odot \left( 1^\omega \land \bigoplus \{ exit(T_i) \mid T_i \in B \} \right).
\]

The series \(tick(B)\) denotes the timing of concurrent execution, where the control flow pauses in the box. The coefficients are computed by summing up the maximum execution time of all the threads. Here, the virtual EOT nodes are included in the calculation, hence the maximum execution time of a thread is \(tick(T_i) \oplus tick(N_{\text{EOT}}(T_i))\). This is because if the control flow pauses in the box (i.e., no joining), reaching an end node is equivalent to pausing at an EOT node.

The series \(exit(B)\) denotes the timing of joining, where the control flows from the concurrent threads merge and exit the box. This formulation has two parts. The first part is the computation of the cost \(\bigcirc tick(N_{\text{EOT}}(T_i))\). A joining takes place when all the threads have terminated, hence the joining cost is computed by summing up the costs for reaching the virtual EOT nodes (i.e., terminated states). If any thread is not yet terminated, that is, \([X^n]tick(N_{\text{EOT}}(T_i)) = 0\), the joining cost \([X^n]exit(B)\) is 0. The second part of the formulation is a condition for joining: \(1^\omega \land \bigoplus exit(T_i)\). A joining takes place when the last child thread terminates, where at least one thread should be able to exit in that tick \((|X^n]exit(T_i) \geq 1\). The term \(\bigoplus exit(T_i)\) is 0 if no thread can exit. This condition prevents a join from occurring when all the virtual EOT nodes are self-looping but no thread can reach the end node. This second part is a boolean series, where the \(\land\) operator and \(1^\omega\) reduce the coefficients to 1 or 0. Therefore, the second part does not contribute to the WCRT. Finally, we compose the two parts using the \(\bigodot\) operator so that \([X^n]exit(B) = 0\) if either part is 0.

As an example, here is the equation system that describes the TCA of Box\text{fork} in Fig. 4.6:

\[
tick(\text{Box}_\text{fork}) = (tick(T_1) \oplus tick(T_{1\text{EOT}})) \bigcirc (tick(T_2) \oplus tick(T_{2\text{EOT}}))
\]

\[
exit(\text{Box}_\text{fork}) = tick(T_{1\text{EOT}}) \bigcirc tick(T_{2\text{EOT}}) \odot (1^\omega \land (exit(T_1) \oplus exit(T_2))).
\]

### 4.3.6 Modeling the TCA of an abort box

The TCA of an abort box \(B\) is defined as follows:

\[
tick(B) = tick(T_{\text{high}}) \bigcirc tick(T_{\text{low}})
\]

\[
exit(B) = exit(T_{\text{high}}) \oplus (tick(T_{\text{high}}) \bigodot exit(T_{\text{low}})).
\]

The series \(tick(B)\) captures the timing when no preemption occurs. It sums up the execution time of \(T_{\text{high}}\) and \(T_{\text{low}}\) as they execute concurrently. If either thread terminates in a tick (i.e., \([X^n]tick(T_{\text{high}}) = 0\) or \([X^n]tick(T_{\text{low}}) = 0\), this triggers a preemption and \([X^n]tick(B) = 0\).
The series \( \text{exit}(B) \) obtains the timing when preemption occurs. \( T_{\text{high}} \) and \( T_{\text{low}} \) trigger a preemption differently. When \( T_{\text{high}} \) triggers a preemption, the exit cost of the box is the exit cost of \( T_{\text{high}} \) (i.e., \( \text{exit}(T_{\text{high}}) \)), since \( T_{\text{low}} \) is preempted. However, when \( T_{\text{low}} \) triggers a preemption, it implies \( T_{\text{high}} \) had executed and reached an EOT node, as \( T_{\text{high}} \) has a higher execution priority. Thus, the exit cost in this case is \( \text{tick}(T_{\text{high}}) \odot \text{exit}(T_{\text{low}}) \). Overall, the exit cost of the box is the maximum of the two cases.

As an example, here is the equation system that describes the TCA of \( \text{Box}_{\text{abort}} \). Note that the WCRT analysis has first flattened the \( \text{ABody} \) thread by substituting the box \( \text{Box}_{\text{fork}} \) with a WCRT equivalent graph derived from its TCA.

\[
\begin{align*}
\text{tick}(\text{Box}_{\text{abort}}) &= \text{tick}(\text{CheckA}) \odot \text{tick}(\text{ABody}) \\
\text{exit}(\text{Box}_{\text{abort}}) &= \text{exit}(\text{CheckA}) \oplus (\text{tick}(\text{CheckA}) \odot \text{exit}(\text{ABody})).
\end{align*}
\]

### 4.4 Implementations

The presented formulations are the theory of WCRT algebra. To realise them as a WCRT analysis technique, we have implemented the logic and the algebra in Python. In this section, we will present the details of this implementation.

The Python implementation has two main parts. The first part is a recursive algorithm that traverses through the hierarchy of the program. This part is the backbone of the implementation, which drives the overall WCRT analysis. Each time it encounters a box, it invokes the second part to construct the TCA, and from which the replacement WCRT equivalent sequential graph is generated.

The second part is the transformation between TCCFG and TCA. Transforming a TCA into a WCRT equivalent graph is straightforward since the TCA itself is similar to a TCCFG of a single thread. The focus of the second part is generating the TCA for a box, which is done using an algebra generator. Given a box, the algebra generator creates the algebra system using the presented formulations; and from this algebra system, it generates a Python program. Then the TCA constructed by repeatedly invoking the generated program.

In the following subsections, we will present these two main parts in more details.
4.4 Implementations

4.4.1 The WCRT analysis logic

Algorithm 1 The backbone algorithm of WCRT algebra

```
1: procedure WCRT_solve(Box)
2:   for Every thread T in Box do
3:     if T has boxes then
4:       for Every b in boxes do
5:         TCA = WCRT_solve(b)
6:         Convert TCA to a WCRT equivalent graph
7:         Replace b with the WCRT equivalent graph
8:       end for
9:     end if
10:   end for
11:   Generate a Python program for Box based on the WCRT algebra
12:   Launch Python program to construct the TCA for Box
13:   if Box consists only the main thread then
14:     Return the maximum timing of TCA (i.e., WCRT)
15:   else
16:     Return the TCA of Box
17:   end if
18: end procedure
```

Let us start from the top level of the implementation. Algorithm 1 shows the pseudo-code that drives the WCRT analysis. Initially, we take the main thread as the input of the algorithm: a box with a single thread. The algorithm first analyses for boxes, and recursively invokes itself for each sub-box found in the threads (Lines 2 to 10). This exploration repeats until it reaches the lowest level of hierarchy which has no boxes, then it invokes the algebra generator to construct the TCA (Lines 11 and 12). Finally, the TCA itself is returned (Lines 13 to 17) and will be used for substituting the original box; the WCRT is returned after the TCA of the main thread is constructed.
4.4.2 Part2: Converting a TCA to a WCRT equivalent graph

Fig. 4.8 presents an illustration of converting a TCA into a WCRT equivalent sequential TCCFG. The process is simple, each state in the TCA is mapped into an EOT and a computation node. In each tick, the execution of the TCCFG has two choices, it can either reach the next EOT node and pause, or exit through the computation node. The execution costs of the nodes are directly extracted from the TCA edges.
4.4 Implementations

4.4.3 Part2: Generating the TCA for a box

![Diagram showing the process of generating the TCA for a box]

When Algorithm 1 discovers a box, this second part is invoked to construct the TCA for the box. The overview of this second part is shown in Fig. 4.9. The first step is to generate the Python program that captures the algebra of the input box, then we invoke the Python program to generate the TCA.

The algebra system of a box is a set of algebraic functions that computes the timing of the program, and we can implement each of these algebraic function as a program function (i.e., taking inputs and producing outputs); and thus, we can create a standalone executable program from the algebra system. Given a box \( B \), we use the proposed formulations (Sec. 4.3) to generate an algebra system, which is a set of algebraic functions \( \mathcal{F} = \bigcup \text{tick}(\Phi) \cup \text{exit}(\Phi) \), where \( \Phi \) is the union of all the nodes and threads in the box \( B \), and the box itself. Each algebraic function \( f(X) \in \mathcal{F} \) is implemented as a Python function, and the idea is to derive the coefficients of the formal power series tick by tick. That is, given the initial conditions (e.g., \([X^0]\text{pre}(f_i(X)) = 0, f_i(X) \in \mathcal{F})\), we compute the 1st coefficient for all the al-
gebraic function (i.e., \([X^nf_i(X), f_i(X) \in \mathcal{F})\). Then we record the results and construct the first state of the TCA. After that we compute the initial conditions for the next tick (i.e., \([X^{T+1}]pre(f_i(X)), f_i(X) \in \mathcal{F}\)), and repeat the process to compute the coefficients for the next tick. The computation stops when the TCA ends, or when it loops back to one of the previous states.

This term-by-term implementation is possible because all the algebraic functions are term-by-term operations, where a coefficient is computed using only the coefficients with the same polynomial power. That is \([X^n]f_i(X), f_i(X) \in \mathcal{F}\) only depends only on \([X^n]f_j(X), f_j(X) \in \mathcal{F}'\), \(\mathcal{F}' \subseteq \mathcal{F}\). Moreover, the initial conditions for the next tick, can be immediately computed from the current coefficients.

In the main function, these Python functions are called in topological order based on their dependency, such that each function is called exactly once to create a TCA state. This is a limitation of Python, where we cannot do forward declarations. For other programming languages, this order is sorted automatically by the compiler.

```python
def B3_tick(tick_dict, exit_dict):
    # tick(B3) = 0
    tick = None
    tick_dict["B3"] = tick

def B3_exit(tick_dict, exit_dict):
    # exit(B3) = \(C_{B3} \ominus (exit(CheckA_{start})) \oplus (exit(B4))\)
    exit = exit_dict.get("CheckA_start")
    exit = my_plus(exit, exit_dict.get("B4"))
    exit = my_dot(exit, 25)
    exit_dict["B3"] = exit

Figure 4.10: Python code example for the computation node B3.
```

Fig. 4.10 shows the python functions for B3 in the running example. Natural numbers in \(\mathbb{N}_\infty\) of the min-max-plus algebra are part of the Python build-in types; the element 0 is denoted using None. The \(\oplus, \land, \ominus\) binary operators are implemented as custom functions my_plus(), my_min() and my_dot() to handle the operations containing 0. The tick and exit costs of each element for the current tick (e.g., \([X^n]tick(B3)\) and \([X^n]exit(B3)\) are stored in the dictionaries tick_dict and exit_dict, and they are shared among all the functions.

### 4.5 Evaluation

In this section, we present an experimental evaluation of WCRT algebra through benchmarking, and compare it with the state-of-the-art model checking based [58] and our ILP\(_C\) (Chapter 3). The benchmarking was conducted in two phases. The first phase evaluates performance using a set of PRET-C programs taken from [58, 72, 73]. The second phase examines the theoretical properties of WCRT algebra using a set of synthetic TCCFGs.
4.5 Evaluation

Since all the Integer Linear Programming (ILP) based techniques and WCRT algebra are unable to perform value-tracking like the model checking and reachability-based techniques, we have abstracted all the data-flow information from the benchmark programs. This will allow a fair comparison for the scalability, as all the techniques will reach the exactly same WCRT for each benchmark program (i.e., same level of precision).

The abstraction of data-flow has two impacts. First, the computed WCRT is less precise compared to when data-flow is considered. Second, the search space for explicit path numeration techniques (i.e., model checking, reachability and WCRT algebra) will be significantly smaller, which benefits their scalability.

4.5.1 Benchmark settings

The benchmarking was carried out on a Windows based computer, with an i5-6300U processor and has 8GB of RAM. Both of the model checking based and ILP\textsubscript{C} used pre-existing tool implementations that could be adapted to TCCFG without modification to the algorithms. The benchmarking process is to apply all three techniques over the same set of TCCFGs, and record the computed WCRTs and their analysis times. In the sequel, we refer to our timing simulation in WCRT algebra as ‘WA’, and to the model checking approach as ‘MC’.

Model checking based approach

The model checking based approach (MC) is based on UPPAAL [79]. The technique first transforms the TCCFG into a functionally equivalent UPPAAL model, with an additional variable to track the execution time. This variable is incremented each time a transition takes place and is reset when advancing to the next tick. The WCRT is computed through a query on the maximum value of the variable.

MC is a more capable technique than the other two in the sense that it can optionally track variable values at the expense of a longer analysis time. For a meaningful comparison, in this benchmarking, we configure the model checking approach to analyse for tick alignment only.

4.5.2 Existing benchmark programs

The details of the benchmark programs and the analysis results from the three techniques are summarised in Table 4.1. The benchmark programs were used in Chapter 3 to benchmark ILP\textsubscript{C}, which cover a range from small to large problem sizes. For example, the largest program is WaterMonitor which has 3204 lines of C code generated from the PRET-C specification consisting of 40 threads. All the techniques produce the same WCRT estimate, which cross-checks the correctness of all three techniques.
Table 4.1: Benchmarking results for model checking (MC), ILP\textsubscript{C} and WCRT algebra (WA).

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Thd</th>
<th>Analysis Time (s)</th>
<th>WCRT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MC</td>
<td>ILP\textsubscript{C}</td>
</tr>
<tr>
<td>ChannelProtocol</td>
<td>591</td>
<td>7</td>
<td>2.63</td>
<td>0.10</td>
</tr>
<tr>
<td>Flasher</td>
<td>816</td>
<td>7</td>
<td>3.22</td>
<td>0.11</td>
</tr>
<tr>
<td>RobotSonar</td>
<td>962</td>
<td>7</td>
<td>6.51</td>
<td>0.63</td>
</tr>
<tr>
<td>Synthetic 1</td>
<td>1287</td>
<td>7</td>
<td>12.00</td>
<td>4.55</td>
</tr>
<tr>
<td>Synthetic 2</td>
<td>1293</td>
<td>7</td>
<td>12.6</td>
<td>2.10</td>
</tr>
<tr>
<td>DrillStation</td>
<td>1094</td>
<td>15</td>
<td>5.30</td>
<td>2.80</td>
</tr>
<tr>
<td>CruiseControl</td>
<td>2302</td>
<td>25</td>
<td>N/A</td>
<td>0.72</td>
</tr>
<tr>
<td>RailroadCrossing</td>
<td>2713</td>
<td>30</td>
<td>N/A</td>
<td>1.05</td>
</tr>
<tr>
<td>WaterMonitor</td>
<td>3204</td>
<td>40</td>
<td>N/A</td>
<td>0.65</td>
</tr>
</tbody>
</table>

The analysis time of MC increases exponentially as the number of program states increases. Eventually, it is not able to finish the analysis for the three largest programs after two minutes due to insufficient memory. This is a typical behaviour for explicit path enumeration, which is the reason that the approach is deemed to be unsuitable for timing analysis. In comparison, ILP\textsubscript{C}, which is based on implicit path enumeration, is much more scalable. On average, ILP\textsubscript{C} only takes 1.41 seconds, with a peak of 4.55 seconds.

However, WA is even faster than ILP\textsubscript{C}, which breaks the conventional belief about timing analysis: explicit path enumeration does not scale well. The average analysis time of WA is 0.4 second, which is 3.5 times faster than ILP\textsubscript{C}. Moreover, the current implementation of WA involves a disk access delay as it writes programs to disk, and then reads them to execute. If we exclude the disk access delay, the actual analysis time of WA can be considered instantaneous.

The prime factor for the short analysis time is the number of TCA states generated. The total number of accumulated TCA states, that is the sum of all TCA states for boxes and the main thread, is shown in parenthesis after the analysis time of WA. We observe that these accumulated TCA states are very small, some of them being even less than the number of threads in the program (i.e., less than one state per thread). This indicates that for realistic synchronous program structures the state explosion problem does not occur and that WA is able to benefit from this well-behaved structure. On the other hand, MC, which generated millions of states, though working at the same abstraction level, runs into a (algorithm-induced) combinatorial explosion.

### 4.5.3 Theoretical properties

WCRT algebra has demonstrated exceptional performance on our benchmark programs, and we believe this is because the state explosion is mitigated by the synchronous composition of TCAs.

To evaluate this hypothesis, we benchmarked the techniques using a set of highly symmetrical
TCCFGs shown in Fig. 4.11. The number of threads is incremented by replicating $T_{1a}$ and $T_{1b}$ alternatively. This set of synthetic TCCFGs was taken from Chapter 3, which was originally designed to stress $ILP_C$.

The results for the first set of synthetic benchmarks are shown in Fig. 4.13. As expected, WCRT algebra only has to generate three TCA states to compute the WCRT regardless of the number of threads. Thus, it results in an analysis time constantly below 0.1 second. In contrast, MC exhibits the same exponential trend as before, and $ILP_C$ also increases exponentially in this case.

On the other hand, how can we see combinatorial explosion in our timing simulation in WCRT algebra? This happens in cases where the concurrent TCAs are highly decoupled in terms of tick alignment. The corner case that triggers the worst-case scenario for WA is shown in Fig. 4.12. The synthetic TCCFGs have an increasing number of threads, and each thread is a sequence of EOT nodes executing in a loop. The key here is the number of EOT nodes, which are prime numbers, starting from 2, then 3, 5 and so on. The total TCA states generated by simulation is the product of the prime numbers, which is the same as the state-space considered.
The results of this second phase is shown in Fig. 4.14. The analysis times of WA and MC grow exponentially as the number of threads (i.e., program states) increases. However, WA is significantly worse than MC. WA is only able to analyse up to five threads, while MC can analyse up to eight threads. WA may seem faster for the first three benchmarks, but the differences are negligible. This difference in performance is a result of the implementation. The Python implementation of WA is much slower than the native binary of the UPPAAL model checker used in MC. Also, we believe the Python implementation is less optimised. On the other hand, ILP$_C$ exhibits the typical behaviour of implicit path numeration, which scales well with respect to program states.

Figure 4.12: Synthetic B.
4.5 Evaluation

Figure 4.13: Results of the model checking, ILP sub C and timing simulation in WCRT algebra with Synthetic A.

Figure 4.14: Results of the model checking, ILP sub C and timing simulation in WCRT algebra with Synthetic B.
4.6 Conclusions

Conventional experience is that explicit path enumeration gives precise results but scales poorly as the number of concurrent threads increases. Thus, it is rarely used for WCRT analysis of synchronous programs. The problem is that the existing approach tries to compute the WCRT, a non-functional property, based on functionally equivalent automata. In this chapter, we presented a new analysis via pairs of formal power series in min-max-plus algebra, called TCAs, which is based on the idea of timing equivalence. We simplify threads as WCRT equivalent TCAs before composing them, which greatly reduces the search space. We could show empirically that WCRT algebra, on the given benchmark examples, is much faster than the most widely used existing approaches.

However, the WCRT algebra presented in this chapter has two notable limitations. First, the efficient composition of concurrent threads relies on abstracting signals. Signals are a common mechanism used in many synchronous languages for communication between threads. Most existing explicit path enumeration techniques are capable of modelling signals. At this stage, the proposed technique falls short in this regard compared to generic model checking. However, even with signal abstraction, the tick alignment problem is still challenging enough for the existing technique to achieve practical analysis time, as shown in our benchmarking and in Chapter 3.

The second limitation is that the proposed technique does not solved the state explosion problem for all cases. As illustrated by our benchmarks (Sec. 4.5.3), if the TCAs cycle through a prime number of states, their composition is the multiplication of the prime numbers, which will result in the same number of operations as the existing techniques. However, we believe such cases are rare in normal programs, thus we can benefit from the abstraction most of the time.
Synchronous paradigm with DVFS

Twenty-five years ago, the synchronous paradigm [6] was proposed to meet the design needs of safety critical reactive systems [17, 80]. The most challenging part was to ensure correct functionalities and timing. The synchronous paradigm is grounded on formal semantics which allows both properties to be formally verified. In other words, the resultant system is guaranteed to always perform the right action at the right time. This correct-by-construction nature is favourable in designing safety critical systems, which separates the synchronous paradigm from all other design approaches. As of today, the synchronous paradigm is still a top choice for safety critical systems [11, 12]. However, as technology progresses, the synchronous paradigm starts to fall short in taking advantage of newer technologies, and one of these technologies is power management.

Power management can benefit all battery powered systems, including some safety critical systems such as a pacemaker [81], by prolonging their battery life. This power consumption aspect has gradually become an important measure in general, as more computer systems are being used in a portable manner. The processor is usually one of the most power hungry component in a system, and many technologies have been developed to reduce its power consumption. Some of the most notable ones include Dynamic Voltage and Frequency Scaling (DVFS), clock gating and power gating. These technologies can be commonly found in modern processors, and they have been a hot research topic for real-time systems. There are numerous algorithms available to utilise these power saving technologies [62, 63, 64, 67, 68]. They are all based on the principle of slack reclaiming, which is to slow down the processor based on the workload, while ensuring the deadline is still met.

While it would be intriguing to use one of the existing power management algorithms with the synchronous paradigm, as it stands, they are not compatible with one another. All the existing
algorithms are specifically developed for Real-Time Operating Systems (RTOSes). They require tight coupling with the scheduler, thus they cannot be applied to synchronous programs which take a bare-metal approach (i.e., no scheduler). Moreover, existing power management algorithms make scheduling decisions on the fly, which conflicts with the proposition of the synchronous paradigm. The effect of the existing algorithms can only be evaluated through simulation due to the dynamicity in scheduling. The result may be an indication of one of the average cases, but not a reliable one. Any change in the input set can dramatically vary the simulation outcome. In contrast, the synchronous paradigm is designed to be on the predictable side, where interleaving of threads is determined during the compile time. The properties of the system, such as timing and functionality, can be analysed statically (i.e., before execution). Thus, it makes sense in the context of the synchronous paradigm that the energy consumption should also be statically analysable.

In this chapter, we present our approach to combining DVFS and the synchronous paradigm. To the best of our knowledge, this is the first attempt to introduce power management for the synchronous paradigm. The essence of using DVFS is to determine where to change the processor frequency (i.e., DVFS control point allocation) and what frequency the processor should change to (i.e., values of the DVFS control points). To preserve the predictability of the synchronous paradigm, we propose a compile time DVFS scheme where the DVFS control point allocation and the frequency values are determined at compile time. To complete the scene, we propose a power measure for the synchronous programs called Worst-Case Energy Consumption (WCEC), which is based on the same principle as the existing measure for timing: Worst-Case Reaction Time (WCRT). WCEC and WCRT are two antagonistic aspects of the system. Increasing the processor frequency can reduce WCRT but results in higher WCEC, and vice versa. In this bi-criteria context, there is no so-called optimal setting. Thus, we developed a bi-criteria analysis technique based on the concept of Pareto optima [82] to explore options with various WCRT and WCEC tradeoffs.

The main contributions of this chapter are as follows:

- We develop a statically analysable DVFS scheme for the synchronous language Precision Timed C (PRET-C). Unlike all the existing algorithms for DVFS where the computation of processor frequency is a black box to the user, our DVFS scheme tightly couples with the paradigm, and allows the user to partially or fully take control of the processor frequency within the synchronous program. This allows the user insights (e.g., assigning lower frequency to the most frequently executed part) to contribute toward the final scheme.

- Given a program with an incomplete frequency specification from the user, we develop a bi-criteria optimisation technique to analyse the remaining frequency options with respect to WCRT and WCEC tradeoffs, and provide multiple schemes for the user to choose from. The biggest challenge here is the huge search space. Given \( n \) frequency control points and \( m \) processor frequencies, the total number of choices are \( n^m \). To overcome the problem, we develop an iterative analysis that consists of two interacting processes: WCRT analysis and the DVFS algorithm. The WCRT analysis provides the WCRT estimation and feedback on the critical path, and the DVFS algorithm is in charge of adjusting the frequencies. By also using a greedy heuristic, we have achieved a scalable performance.
• We evaluate the proposed technique using benchmark programs taken from [73], and produce the estimated Pareto front in the WCRT vs. WCEC space. We compare the results with the conventional approach where a single frequency is used, and an existing linearised approach. The results show our approach produce more non-dominant choices to the user than the single frequency approach and the linearised approach, while it is also safer than the linearised approach which may miss the WCRT constraint.

This chapter is organised as follows: We start by introducing the preliminaries for the chapter in Sec. 5.1. Then we present the proposed DVFS scheme for PRET-C in Sec. 5.2. After that we present the overview of the bi-criteria optimisation algorithm in Sec. 5.3. Subsequently, the interactive steps of the optimisation algorithm: the WCRT/WCEC analysis and the DVFS algorithm, are detailed in Section 5.4 and Section 5.5 respectively. Finally, we present our experimental results in the Sec. 5.6, and the chapter concludes in Sec. 5.7.

5.1 Preliminaries

5.1.1 Running example

```c
void T1()
{
    EOT;
    foo1();
}

void T2()
{
    EOT;
    foo2();
}

void main()
{
    if (X)
    {
        PAR(T1, T2);
    }
    else
    {
        EOT;
        foo3();
    }
}
```

Figure 5.1: The PRET-C code of the running example.

Here, we present a running example that will be used to demonstrate the proposed DVFS scheme and bi-criteria optimisation techniques. The PRET-C code of the running example is
shown in Fig. 5.1, and its Timed Concurrent Control Flow Graph (TCCFG) representation is shown in Fig. 5.2.

The execution of this running example is simple. In the first tick, the program starts from the start node $B_0$, and takes one of the branches at the condition node $B_1$ based on the input $X$. If the branch $E_2$ is taken, the fork node $B_2$ executes and spawns the child threads $T_1$ and $T_2$. Then the program pauses at the EOT nodes $B_3$ and $B_5$. If the other branch $E_{10}$ is taken, the program pauses at the EOT node $B_7$. In the second tick, depending on which branch is taken in the first tick, the program either resumes from $B_3$ and $B_5$ and executes the computation nodes $B_4$ and $B_6$ concurrently, or resumes from $B_7$ and executes the computation node $B_8$. Either way, the program eventually reaches the end node and finishes.

A fragment of the TCCFG is outlined in Fig. 5.2 with the computation nodes $B_4$, $B_6$, and $B_8$ inside. Let us assume that the execution time and energy consumption of all other nodes are negligible in this running example. Then we only need to consider the execution in the second tick where the program executes either $B_4$ and $B_6$ concurrently, or executes $B_8$. The execution costs of $B_4$, $B_6$ and $B_8$ are assumed to be 90, 30 and 110 processor cycles respectively.

5.1.2 Worst-Case Energy Consumption

The timing of a synchronous program is measured using WCRT, which is the longest possible execution time of a tick. In this work, we apply the same principle for measuring the power consumption, that is the largest possible energy consumption of a tick. We call this the Worst-Case Energy Consumption (WCEC). The motivation is to derive, for instance, a lower bound on the life of a battery powered embedded system.

5.1.3 Dynamic Voltage and Frequency Scaling

The power consumption of a processor consists of two components, namely static and dynamic power consumption respectively. Static power consumption is caused by leakage current of transistors, which persists as long as the processor is switched on. Dynamic power consumption, on the other hand, is dependent on the voltage and frequency of the processor. Considering that synchronous programs are hard to be distributed and are mostly only compiled for single-core processors [83], powering off the processor is not an option (e.g., clock gating and power gating), in this work we focus on reducing the dynamic power consumption.

DVFS is a well-known technology for reducing dynamic power consumption, which is achieved by changing the operating point of the processor, where an operating point is a frequency and voltage pair. The strategy to change the operating point is known as a DVFS scheme. For this work, our target processor is MicroBlaze, which we assume is manufactured using 90 nm technology [84] and has the characteristics shown in Table 5.1.
The process can operate at four frequencies: 25 MHz, 50 MHz, 75 MHz and 100 MHz, and each frequency is paired with a voltage that allows stable operation. The static and dynamic power consumption is estimated using the following well-known formula [84, 85]:

\[
P_{\text{dynamic}} = \left( \frac{1}{2} \times C \times V^2 + E \right) \times f
\]

\[
P_{\text{static}} = V \times I_{\text{static}}
\]

We assume the processor uses 600 cells (estimated based on the synthesised logic elements on FPGA), and the constants \( C, E, \) and \( I_{\text{static}} \) are typical values in production taken from the manufacturer’s handbook [84]. The total power consumption of the processor at a specific frequency is the sum of the static and dynamic power consumptions.

Given the processor frequencies and their power consumptions, we can calculate the energy consumption of each clock cycle as follows:

\[
\text{Energy per cycle} = P_{\text{total}} \times \text{cycle length}
\]

These energy-per-cycle values will be used for computing the energy consumption for executing a node in the TCCFG, which is subsequently used for computing the WCEC of the program. For our target processor, the energy-per-cycle information is summarised in Table 5.2 below.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>25 MHz</th>
<th>50 MHz</th>
<th>75 MHz</th>
<th>100 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy per cycle</td>
<td>0.064</td>
<td>0.073</td>
<td>0.084</td>
<td>0.099</td>
</tr>
</tbody>
</table>

Table 5.2: The energy consumption per clock cycle of the target processor (Unit: \( \mu \)).

### 5.1.4 Pareto Optima

Given a DVFS scheme, we can classify it using its WCRT and WCEC values, and plot it in a (WCRT, WCEC) space. Fig. 5.3 shows an example of the (WCRT, WCEC) space with one point \((x, y)\) (i.e., a DVFS scheme). In the (WCRT, WCEC) space, we can compare different DVFS schemes using the notion of dominance and Pareto optima. The idea of dominance is to compare two points in space based on their relative position. For example, the point \((x', y')\) divides the space into four panes as shown in Fig. 5.3, and its relation to the point \((x, y)\) is defined as follows:
Synchronous paradigm with DVFS

The synchronous paradigm is known for its predictability, which allows static analysis of the system properties. Following this convention, we propose a compile time DVFS scheme that tightly couples the synchronous paradigm.

5.2 DVFS schemes for synchronous programs

The definitions of Pareto optima are as follows:

- A point is a weak Pareto optimum iff there does not exist another point that strongly dominates it (but there might exist other points that weakly dominate it).
- A point is a strong Pareto optimum iff there does not exist another point that (weakly or strongly) dominates it.
- The Pareto front is the set of all the weak and strong Pareto optima.

Given a PRET-C program, our goal is to construct the Pareto front and let the user choose from one of the tradeoffs.
A compile time DVFS scheme is a set of DVFS control points that are statically inserted in
the program, and each is assigned with a frequency value. To make our DVFS scheme more
generic, these control points should make use of the common notions in the synchronous
paradigm. In our case, we choose to insert these control points in the tick barriers which mark
the context switching points. In PRET-C, these are the start node, EOT nodes and join nodes.
For example, in the running example in Fig. 5.2, a DVFS control point is inserted in the start
node $B_0$, EOT nodes $B_3$, $B_5$ and $B_7$, and the join node $B_9$.

```c
1  ReactiveInput(int, X, 0);
2  void T1(){
3      EOT; //@f1
4      foo1();
5  }
6
7  void T2(){
8      EOT; //@f1
9      foo2();
10 }
11
12 void main(){
13    if (X){
14      PAR(T1, T2); //@f2
15    }else{
16      EOT;
17      foo3();
18    }
19 }
```

Figure 5.4: The PRET-C running example with frequency annotations.

Since these points are well-defined in the synchronous paradigm, they are part of the source
code. This allows the user to partially, or fully, control the processor frequency within the
program. For example, Fig. 5.4 shows the PRET-C code of the running example with frequency
annotations. The EOT and PAR statements in line 3, 8 and 14 are annotated with frequencies,
marking $B_3$ and $B_5$ for switching the frequency to $f_1$, and $B_9$ for switching to $f_2$. This
information can then be taken into account during compilation. The user can also choose not
to annotate (e.g., line 16), and let the bicriteria algorithm presented in Sec. 5.3 explore for
options.

### 5.2.1 The running example with DVFS scheme

Given a PRET-C program with $n$ start, EOT and join nodes, and a processor with $m$ operating
frequencies, there are total of $m^n$ possible DVFS schemes. In this section, we illustrate how
varying the frequencies affects the execution time and energy consumption of a node using
the running example in Fig 5.2.
In the second tick of the execution, the running example can execute either $B_4$, $B_6$ concurrently, or execute $B_8$. These computation nodes may execute at different frequencies, depending on the values of the DVFS control points at the EOT nodes $B_3$, $B_5$ and $B_7$. Consequently, their execution time may also vary. Here, we assume the execution time of a node scales linearly with the processor frequency. For example, if the running example is executing on our target processor (the characteristics are shown in Table 5.1), then the execution times of the computation nodes $B_4$, $B_6$ and $B_8$ are simply their execution costs in processor cycles multiplied by the cycle length. The calculation results are summarised in Table 5.3.

<table>
<thead>
<tr>
<th>Node</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25 MHz</td>
</tr>
<tr>
<td>B4</td>
<td>3.6</td>
</tr>
<tr>
<td>B6</td>
<td>1.2</td>
</tr>
<tr>
<td>B8</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Table 5.3: The execution time of the computation nodes in Fig. 5.2 at four different frequencies (Unit: $\mu$s).

Similarly, the energy consumptions of nodes $B_4$, $B_6$ and $B_8$ are calculated by multiplying their execution costs in processor cycles with energy per cycle. The calculation results are summarised in Table 5.4.

<table>
<thead>
<tr>
<th>Node</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25 MHz</td>
</tr>
<tr>
<td>B4</td>
<td>5.73</td>
</tr>
<tr>
<td>B6</td>
<td>1.91</td>
</tr>
<tr>
<td>B8</td>
<td>7.00</td>
</tr>
</tbody>
</table>

Table 5.4: The energy consumption of the computation nodes in Fig. 5.2 at four different frequencies (Unit: $\mu$J).

### 5.2.2 Comparing DVFS schemes

A DVFS scheme can vary the execution time and energy consumption of a node, which consequently impacts the WCRT and WCEC of this program. It is important that we determine the relative superiority of one scheme over another. We compare these schemes using their WCRT and WCEC and the notion of Pareto dominance.

For the running example, the computations of WCRT and WCEC can be abstracted as max operations. Let $t(n)$ denote the execution time of the node $n$ and $e(n)$ denote its energy consumption. Then, for this running example, the WCRT and WCEC are calculated as:

\[
\text{WCRT} = \max(t(B_8), (t(B_4) + t(B_6)))
\]
\[
\text{WCEC} = \max(e(B_8), (e(B_4) + e(B_6)))
\]
WCRT and WCEC are antagonistic in nature: reducing the WCRT requires us to increase the frequencies, which in turn increases the WCEC. Similarly, reducing the WCEC requires us to decrease the frequencies, which in turn increases the WCRT. Table 5.5 shows four of the DVFS schemes for the running example, with their corresponding WCRT and WCEC. Scheme 2 is clearly worse than scheme 1 as it has the same WCRT but a larger WCEC: in other words, scheme 1 weakly dominates scheme 2. Similarly, scheme 3 weakly dominates scheme 4. Scheme 1 is incomparable to scheme 3, because they cannot be better or equal in both WCRT and WCEC. In these four schemes, either scheme 1 or scheme 3 should be chosen in practice. However, to pick the best scheme based on need, we have to compare all the schemes with respect to their WCRT and WCEC.

Table 5.5: DVFS scheme examples for the running example.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>DVFS values (MHz)</th>
<th>WCRT (µs)</th>
<th>WCEC (µJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f_{B4}=50, f_{B6}=75, f_{B8}=50</td>
<td>2.2</td>
<td>9.06</td>
</tr>
<tr>
<td>2</td>
<td>f_{B4}=75, f_{B6}=50, f_{B8}=50</td>
<td>2.2</td>
<td>9.76</td>
</tr>
<tr>
<td>3</td>
<td>f_{B4}=75, f_{B6}=100, f_{B8}=100</td>
<td>1.5</td>
<td>10.86</td>
</tr>
<tr>
<td>4</td>
<td>f_{B4}=100, f_{B6}=50, f_{B8}=100</td>
<td>1.5</td>
<td>11.06</td>
</tr>
</tbody>
</table>

Considering only the three EOT nodes in the running example, there are 64 possible DVFS schemes. We have plotted them in a (WCRT, WCEC) space as shown in Fig. 5.5. The Pareto front is a collection of all the Pareto optima, which are schemes with different tradeoffs in WCRT and WCEC. These are best options, and in practice the user should always choose a scheme on the Pareto front. For all other schemes that are not on the Pareto front, they are
dominated by at least one scheme on the Pareto front, that is, there exists at least one scheme that is better in both WCRTs and WCEC.

In the next section, we present our approach to estimating the Pareto front for a given PRET-C program.

5.3 Bicriteria optimisation algorithm

We have presented a DVFS scheme for PRET-C in Sec. 5.2, which allows the user to directly control the processor frequency. Given a PRET-C program with a partial or no frequency specification, our goal is to estimate the Pareto front to find a set non-dominated options with various tradeoffs. The biggest challenge here is the exponentially growing search space. In this section, we present our bicriteria optimisation algorithm for estimating the Pareto front.

5.3.1 Overview

The proposed algorithm is based on the well-known $E$-constraint approach [86]. The idea is to turn either WCRT or WCEC into a constraint, and optimise the other criteria under this constraint. For each constraint, one point is produced in the (WCRT, WCEC) space. By progressively moving the constraints, we can effectively scan through and produce points across the (WCRT, WCEC) space.

As an example, the pseudo code of the proposed algorithm is shown in Algorithm 2. In this algorithm, the WCRT is used as the constraint, and we optimise for WCEC. First, we determine the feasible range of the WCRT (line 2), by computing the WCRT of the DVFS schemes at the two ends: where all the DVFS control points are set to the slowest frequency (i.e., maximum WCRT) and fastest frequency (i.e., minimum WCRT). Then we start with using the minimum values as the constraint (line 3), and optimise the WCEC (line 5). This effectively simplifies the bicriteria optimisation into a single criteria optimisation (i.e., WCEC optimisation), since we no longer consider whether the WCRT is optimal. Once we find a solution (line 6), we increase the WCRT constraint with a constant step size $\Delta$. The analysis repeats until the

Algorithm 2 Algorithm for constructing the Pareto front

1: procedure CONSTRUCT_PF(TCCFG)
2: Compute the Maximum and Minimum value for WCRT
3: $WCRT\_constraint = $ Minimum value
4: while $WCRT\_constraint < $ Maximum value do
5: Optimise WCEC subject to WCRT $\leq WCRT\_constraint$
6: Record the frequency values, WCRT and WCEC
7: $WCRT\_constraint += \Delta$
8: end while
9: end procedure
5.3 Bicriteria optimisation algorithm

WCRT constraint reaches the maximum value. At the end of the analysis, we collect all the non-dominated points to form an estimate Pareto front. This algorithm can also use WCEC as the constraint and optimise for the WCRT.

Note that this algorithm is not an exact method (i.e., constructing the actual Pareto front) like the conventional \( \mathcal{E} \)-constraint approach, since we use a constant step size for the constraint. This is a necessary compromise for overcoming the problem of the exponentially growing search space. However, given a small enough step size, we can quite precisely estimate the Pareto front.

5.3.2 Optimisation with constraints

![Diagram showing four frequencies: 100 MHz, 75 MHz, 50 MHz, 25 MHz, with corresponding WCRT values.](image)

Figure 5.6: There are a few known Pareto optima in the (WCRT, WCEC) space. These optima have all the DVFS control points set to a single frequency.

When one of the criteria is turned into a constraint, we can attempt to optimise the other criteria subject to the constraint. We still have to face the huge search space here, and the idea of our approach is to start from one of the known Pareto optima, and adjust one DVFS control point by one frequency step at a time to move toward the constraint.

It is known that the processor speed and power consumption are optimal when the processor uses a single frequency [82]. Therefore, we know that it is a Pareto optimum when all the DVFS control points are set to a single frequency. In our case, there are four known Pareto optima (our target processor has four frequencies) which are shown in Fig. 5.6. Increasing the frequency of the DVFS control points will result in larger WCEC and smaller WCRT, moving towards the top and left of the space; and vice versa, decreasing the frequency will move toward the bottom right.

Given a constraint, there are four possible exploration paths to search for non-dominated points that satisfy the constraint (Fig 5.7). For a WCRT constraint, we need to find a point to the left of the constraint (i.e., WCRT \( \leq \) constraint). Starting from one of the fixed frequency Pareto optima, we can find such a point by performing a forward search (i.e., by lowering the frequencies as shown in Fig. 5.7(a)) or a backward search (i.e., by increasing the frequencies as shown in Fig. 5.7(b)). As shown in Fig. 5.7(a), in the forward searching with a WCRT constraint we track back one step to produce an end-point that satisfies the constraint. Similarly,
for a WCEC constraint (i.e., $\text{WCEC} \leq \text{constraint}$), we can perform a forward or backward search (Fig. 5.7(c) and Fig. 5.7(d)).

For each type of exploration, the maximum/minimum frequency and the starting point are determined by the value of the constraint. Given a WCRT constraint, we take the closest fixed frequency Pareto optimum to the left of the constraint as a reference (Fig. 5.8(a)). In a forward search, we start the exploration from this reference point. In a backward search, we take the frequency of the reference point as the maximum frequency: we start from the right-most fixed frequency Pareto optimum (i.e., slowest frequency) and explore toward the left, but frequencies of the DVFS points are only increased up to the frequency of the reference point, since we know we can satisfy the constraint without using a higher frequency than the reference point. Similarly, for a given WCEC constraint (Fig. 5.8(b)), we use the closest fixed frequency Pareto optimum below the constraint as the reference point, which serves as the starting point for the backward search, and as the minimum frequency for the forward search. As the constraint changes as shown in Algorithm 2, the maximum/minimum frequency and the starting point also change accordingly.

Note that we do not perform backtracking in all of our explorations (i.e., we only move in the direction toward the constraint) to reduce the analysis time. This is a compromise we made to tackle the huge search space problem. There is no guarantee that the resultant end-point is a Pareto optimum. Given a point, we find the next one based on heuristics. However, this is sufficient for our work: to produce non-dominated points between the fixed frequency optima. The goal is to utilise DVFS to provide more options to the user, apart from using a
5.3 Bicriteria optimisation algorithm

(a) Pareto optima with a WCRT constraint.  (b) Pareto optima with a WCEC constraint.

Figure 5.8: Given a constraint and the knowledge of the Pareto optima with fixed frequency, we can determine the starting point of the search and the maximum/minimum frequency that should be used.

Figure 5.9: Details of backward searching with WCRT constraint.

Each step in the four explorations is actually an interaction of two analysis processes: one for computing the critical path and one for adjusting the frequencies. To ensure each adjustment can actually move the point toward the constraint, we need to find the critical path of that criterion. For example, given a WCRT constraint, the best approach is to adjust the DVFS control point on the critical path that produces the WCRT. This guarantees an impact in the WCRT of the program, and by repeating this process, the exploration can eventually reach a point that satisfies the constraint.
As an example, Fig. 5.9 shows the details of a backward search with a WCRT constraint. The search starts with a TCCFG and an initial DVFS scheme where all unspecified DVFS control points are set to the slowest frequency (e.g., 25 MHz for our processor). The maximum frequency that the DVFS points can use is determined by comparing the WCRT constraint and the fixed frequency optima. The analysis starts by computing the WCRT and its corresponding execution path ①. Then the WCRT is compared to the WCRT constraint ②. If the WCRT is larger than the constraint, then the DVFS algorithm will take the corresponding execution path, and increase the frequencies of the DVFS control points along that path to reduce its execution time ③. Then the analysis starts over again with the WCRT analysis ①, to search for the other execution paths that exceed the constraint. As soon as the computed WCRT is smaller than the constraint ②, the analysis finishes: the DVFS scheme along with the WCRT and its corresponding execution path are reported. Finally, the WCEC of the reported scheme is computed ④. If the DVFS algorithm ③ cannot make an execution path shorter than the constraint (i.e., all DVFS control points are already at the highest frequency), then the analysis terminates with an error, stating that the constraint is not achievable. All four types of exploration (i.e., forward and backward, with a WCRT or WCEC constraint) use exactly the same framework shown in Fig. 5.9, but with slight variations in each process to adapt to the need of the search.

In the following sections, we will present more details on the analysis technique developed for computing the WCRT ① and the WCEC ④, as well as the DVFS scheme adjustment ③, which uses a heuristic algorithm to further reduce the analysis time.

### 5.4 WCRT and WCEC analysis

Computations for WCRT and WCEC are similar in nature. They both try to find the execution path that maximise a system property. They both can be computed using the same path analysis technique. The only difference is that WCRT analysis uses execution time as the cost for the node, and WCEC analysis uses energy consumption. Since this path analysis is invoked many times over the course of estimating the Pareto front, it is important that the technique must be fast. ILP\(_C\) proposed in Chapter 3 is an ideal candidate for this part, as it is one of the fastest WCRT analysis techniques and it is based on the extensible technique Integer Linear Programming (ILP).

We propose an extension of the ILP\(_C\) technique for computing the WCRT and WCEC from a given TCCFG, which is DVFS aware. In the following subsections, we first quickly recap the formulation of ILP using the running example in Fig. 5.2, then our extension for taking into account the DVFS scheme. For simplicity, the outflow edge of an EOT node or a start node is termed an EOT edge.
5.4.1 Modeling the high-level control flow

ILP_C is based on the mathematical technique ILP. The objective function of the ILP model is to maximize the sum of all the edges multiplied by their associated costs, which has a general form as follows:

$$\max \sum_{i=1}^{N} E_i \times c_i$$

For our running example, $N = 12$ and $c_i$ denotes either the associate cost of $E_i$, which is execution time in the WCRT analysis, or energy consumption in WCEC analysis.

As for the ILP constraints, two assumptions are made based on the properties of synchronous programs and the nature of ILP. First, the value of each edge is bounded to be either ‘1’ (active) or ‘0’ (inactive). This assumption is valid for any synchronous program as they do not have recursion and instantaneous loops. Second, an edge is set to ‘1’ whenever possible, since the execution/energy costs of all nodes have positive contributions toward the objective function (i.e., all costs are positive values).

Computation and condition nodes

The control flows at computation and condition nodes are modeled as inflow edges equal to the outflow edges, as the number of times the control flow enters a computation or condition node must be equal to the number of times it leaves the node. For example, the following constraints are created for the running example:

- $B1: E1 = E2 + E10$
- $B4: E4 = E5$
- $B6: E7 = E8$
- $B8: E10 = E12$

EOT nodes

The ILP constraints for EOT nodes are used for emulating the execution of one tick and enforce the parent-child relationships. This is achieved by eliminating the EOT edge combinations, which are infeasible. For example, the following ILP constraints are generated for the running example:

- $T0-T1: E1 + E11 + E4 \leq 1$
- $T0-T2: E1 + E11 + E7 \leq 1$
Each ILP constraint consists of a group of EOT edges which are mutually exclusive, i.e., at most one of them can be ‘1’. For example, the constraint $T0-T1$ prevents the threads $T0$ and $T1$ from executing concurrently, since $T0$ is the parent thread of $T1$. Similarly, the other constraints eliminate other unfeasible combinations. Only $T1$ and $T2$ are allowed to execute concurrently.

**Fork nodes**

Fork nodes spawn child threads when the control flow reaches them. Therefore, all the outflow edges of a fork are active if any of its inflow edges are active. The ILP constraints for the fork node $B2$ are as follows:

$$B2: E2 = E3, E2 = E6$$

**Join nodes**

The execution of a join node is denoted by its outflow edge. When the last child thread terminates, the outflow edge of a join node is active; otherwise it should remain inactive. Given that the ILP solver will set the outflow edge to ‘1’ if no ILP constraint is created for the join node, our formulation only needs to model the cases where the outflow edge must be ‘0’. For example, the following ILP constraints are generated for the join node $B9$:

$$\text{LastChild: } E5 + E8 \geq E9$$

$$T1: (1 - E4 - E2) + (E5) \geq E9$$

$$T2: (1 - E7 - E2) + (E8) \geq E9$$

The outflow edge $E12$ is on the right-hand side of all the ILP constraints. It is ‘0’ if any term on the left-hand side of the constraints is ‘0’, otherwise it is set to ‘1’ by the ILP solver. The $\text{LastChild}$ constraint captures the moment when the last child thread reaches the join node $B12$, during which at least one of $E5$ and $E8$ must be active.

The constraints for $T1$ and $T2$ capture the states of the two child threads respectively. A child thread forces the outflow edge of the join node to a value ‘0’ if (i) it has an active edge, and (ii) it does not reach the join node. This is captured as two parts in the left-hand side of the constraints (separated by parentheses). For example, if any edge in $T1$ is active, then either its EOT edge $E4$ is active or the inflow edge of the fork node $E2$ is active, since all the edges in $T1$ are initiated from them. So, the first part $1 - E4 - E2$ is ‘0’ if this is true ($E4$ and $E2$ are mutually exclusive from each other because of the constraints for EOT nodes). The second part $E5$ is ‘0’ if the control does not reach the join node. If any of the two parts is ‘1’, then the outflow edge $E12$ is set to ‘1’ provided that $T2$ also meets the same condition.
Figure 5.10: Converting the TCCFG in Fig. 5.2 into four virtual TCCFGs for emulating DVFS.
5.4.2 Modeling the DVFS scheme

The processor cycle has been the primary unit in conventional timing analysis. The actual physical time is calculated by multiplying the number of clock cycles with the cycle length (which is $1/f$). However, when DVFS is employed, the processor frequency changes during the execution, and this effect must be taken into consideration in the timing analysis. This change of frequency complicates the ILP formulation as it introduces multiple costs for each node. Similarly, the energy consumption of a node also changes depending on the frequency.

Conventional ILP modelling cannot model nodes with variable costs. To cope with this limitation, we consider a TCCFG with DVFS as multiple virtual TCCFGs, where each virtual TCCFG corresponds to one specific frequency of the processor. An illustration of this for our running example TCCFG (Fig. 5.2) is shown in Fig. 5.10. The frequency switching is emulated as the control flow jumping between these virtual TCCFGs.

The virtual TCCFGs are identical to the original, except that the cost of each node is based on the associated frequency. The calculations of execution and energy cost are shown in Section 5.2.1. The timing penalty for switching frequency is modeled as a constant cost added to the start node, and all the EOT and join nodes (i.e., all the DVFS control points).

Now, the virtual TCCFGs introduce multiple avatars for each edge in the original TCCFG. For example, with processor frequencies 25MHz, 50MHz, 75MHz and 100MHz, an edge $E_i$ in the original TCCFG becomes four avatars, one for each frequency. We differentiate these avatars using suffixes, so that the avatars of $E_i$ are $E_i_a$, $E_i_b$, $E_i_c$ and $E_i_d$, correspond to 25MHz, 50MHz, 75MHz and 100MHz respectively. Each avatar, like the original edge, is a binary value.

At any time, only one avatar of an edge can be active in an execution path, since a node can only execute at one frequency. Thus, the sum of all the avatars of an edge should be at most 1. To enforce this, the following ILP constraint is created for each edge:

$$E_i_a + E_i_b + E_i_c + E_i_d \leq 1$$

After creating all the avatars, we rebuild all the ILP constraints for a TCCFG with these avatars. We develop a systematic approach, which can preserve the modelling of the high-level control flow while integrating the DVFS scheme. Each ILP formulation is treated differently depending whether the node has a DVFS control point or not.
5.4.3 Nodes without DVFS control points

For the computation nodes, condition nodes, and fork nodes which do not have frequency control points, we consider each virtual TCCFG independently and generate ILP constraints for each using the corresponding avatars. For example, the following ILP constraints are generated for the running example:

25MHz TCCFG:

\[
\begin{align*}
B1: & \ E1_a = E2_a + E9_a \\
B2: & \ E2_a = E3_a; E2_a = E6_a; \\
B4: & \ E4_a = E5_a
\end{align*}
\]

50 MHz TCCFG:

\[
\begin{align*}
B1: & \ E1_b = E2_b + E9_b \\
B2: & \ E2_b = E3_b; E2_b = E6_b; \\
B4: & \ E4_b = E5_b
\end{align*}
\]

The modeling of the control flow is preserved, as the generated ILP constraints are identical to the original with respect to each virtual TCCFG. However, the processor frequency cannot be changed during the execution of these nodes. For example, if the condition node \( B1 \) executes at 50MHz, then its inflow edge \( E1_b \) is ‘1’ (the other avatars are ‘0’). The ILP constraint will allow \( B1 \) to choose its outflow edge, but the processor frequency must remain to be 50MHz (i.e., the only choices are \( E2_b \) and \( E9_b \)).
5.4.4 Nodes with DVFS control points

For the EOT and join nodes which have the DVFS control points, we consider all the virtual TCCFGs as a whole, and create a combined set of ILP constraints to allow the control flow to jump between them. We replace each edge in the original ILP formulation with the sum of its avatars. For example, the following ILP constraints are generated for our running example:

EOT nodes:

\[
\begin{align*}
T0-T1: \quad & (E1_a + E1_b + E1_c + E1_d) + \\
& (E11_a + E11_b + E11_c + E11_d) + \\
& (E4_a + E4_b + E4_c + E4_d) \leq 1
\end{align*}
\]

\[
\begin{align*}
T0-T2: \quad & (E1_a + E1_b + E1_c + E1_d) + \\
& (E11_a + E11_b + E11_c + E11_d) + \\
& (E7_a + E7_b + E7_c + E7_d) \leq 1
\end{align*}
\]

... 

Join node B9:

\[
\begin{align*}
\text{LastChild:} \quad & (E5_a + E5_b + E5_c + E5_d) + \\
& (E8_a + E8_b + E8_c + E8_d) \geq \\
& (E9_a + E9_b + E9_c + E9_d)
\end{align*}
\]

\[
\begin{align*}
T1: \quad & [1 - (E4_a + E4_b + E4_c + E4_d) + \\
& - (E2_a + E2_b + E2_c + E2_d)] + \\
& (E5_a + E5_b + E5_c + E5_d) \geq \\
& (E9_a + E9_b + E9_c + E9_d)
\end{align*}
\]

\[
\begin{align*}
T2: \quad & [1 - (E7_a + E7_b + E7_c + E7_d) + \\
& - (E2_a + E2_b + E2_c + E2_d)] + \\
& (E8_a + E8_b + E8_c + E8_d) \geq \\
& (E9_a + E9_b + E9_c + E9_d)
\end{align*}
\]

These ILP constraints preserve the modeling of the control flow while allowing the frequency to be changed. For example, if T1 executes at 25MHz (i.e., E4_a and E5_a are ‘1’) and T2 executes at 50MHz (i.e., E7_b and E8_b are ‘1’), then the control flow of the two threads are on different virtual TCCFGs, but the ILP constraints still allow their join node B9 to be active. Furthermore, the join node B9 can switch to any frequency (i.e., it is free to select any avatars of E9).
5.5 The DVFS algorithms

5.4.5 Choosing a frequency

Give the ILP model of the control flow and the DVFS scheme, the ILP solver is free to choose any frequency for the control points. To specify a frequency for a control point, we need to create a set of ILP constraints over the avatars of its outflow edge.

Avatars of an edge have dual role characteristics: they represent the choice of processor frequency, and the control flow of the program. For example, let us consider the join node B9. Its outflow edge E9 is converted to four avatars E9_a, E9_b, E9_c and E9_d in the formulation, which denote switching the processor frequency to 25 MHz, 50 MHz, 75 MHz and 100 MHz respectively. At the same time, they denote the execution of the join node (i.e., the control flow resumes into the main thread). Setting an avatar of E9 to ‘1’ forces the solver to use a specific frequency; however, at the same time it also forces the solver to consider only the execution paths that execute the join node. This can result in an under-estimation. Thus, we use an alternative approach that sets all the unused frequencies to ‘0’. This simply stops the ILP solver using these frequencies (i.e., avatars) without side effects on the control flow modelling. To choose a specific frequency for the control point at the join node B9, we create one of the following sets of ILP constraints:

- **25 MHz**
  - \( E9_b = 0 \)
  - \( E9_c = 0 \)
  - \( E9_d = 0 \)

- **50 MHz**
  - \( E9_a = 0 \)
  - \( E9_c = 0 \)
  - \( E9_d = 0 \)

- **75 MHz**
  - \( E9_a = 0 \)
  - \( E9_b = 0 \)
  - \( E9_d = 0 \)

- **100 MHz**
  - \( E9_a = 0 \)
  - \( E9_b = 0 \)
  - \( E9_c = 0 \)

Updating the frequency of a control point is simply changing the ILP constraints from one set to another.

5.5 The DVFS algorithms

If a DVFS scheme cannot satisfy the constraint, the DVFS algorithm is invoked, which takes the critical path of the system and adjusts the DVFS control points along the path. We propose a greedy heuristic for adjusting these frequencies. The idea of the heuristic is to move the point toward the constraint as much as possible in each frequency adjustment. This greedy approach is used for the four types of searches presented in Sec. 5.3.2.

To illustrate this greedy heuristic, let us consider the scenario for backward searching with a WCRT constraint. Fig. 5.11 shows the overview of the DVFS algorithm. It takes the WCRT constraint, the critical path for WCRT and the maximum frequency as inputs. The maximum frequency is derived by comparing the WCRT constraint with the fixed frequency optima (see Sec. 5.3.2). The DVFS algorithm first increases the frequency of a DVFS control point in the critical path: it adjusts exactly one DVFS control point with one frequency step, and prioritizes the DVFS control points which yield the largest reduction in execution time (i.e., the largest possible move toward the constraint). The frequency of each control point can only...
be increased up to the specified maximum value. After that the execution time of the input path is re-computed using the new frequencies and compared with the WCRT constraint. This process repeats until the execution time of the input path is less than the WCRT constraint, and the frequency changes are outputted.

5.6 Results

In this section, we present our evaluation of the proposed DVFS scheme and the bicriteria optimisation technique. The goal of this work is to utilise the DVFS to provide more options with different WCRT and WCEC tradeoffs to the user. We first evaluate this goal by computing the Pareto fronts of a set of industrial applications, using the four types of explorations presented in Sec. 5.3.2. We plot the results in the (WCRT, WCEC) space to observe how the generated schemes distribute across the space. Then we evaluate the optimality of the generated schemes by comparing to two existing approaches: the fixed frequency approach, which uses a single processor frequency throughout the execution, and the linearised approach [65, 66]. Finally, we discuss the practicality of this work in terms of analysis time.
5.6 Results

5.6.1 Benchmark method

The target platform is the MicroBlaze processor [74], which is assumed to be produced using 90 nm technology. The processor frequencies and power consumption are detailed in Sec. 5.1.3. The timing penalty for switching the processor frequency is assumed to be 100 ns. The benchmarking was performed in a Windows based machine with an Intel i7-4650U processor and 8 GB of RAM.

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Threads</th>
<th>DVFS control points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Protocol</td>
<td>591</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>Flasher</td>
<td>816</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>962</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>Cruise Controller</td>
<td>2302</td>
<td>25</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 5.6: Information of the benchmark programs.

The industrial applications used for benchmarking are taken from [73]. For each program, we first build the Concurrent Control Flow Graph (CCFG), then we compile the programs into C code and subsequently into binary to obtain the exact number of processor cycles of each node, therefore resulting in the TCCFG (that is, the Timed CCFG). The details of the programs are summarized in Table 5.6. The columns, from left to right, are the names of the programs, the number of lines of generated C code, number of threads in the original synchronous program and number of DVFS control points in the program. The largest program is the Cruise Controller, which generates 2302 lines of C code, has 25 threads, and 60 DVFS control points.

The proposed technique

We apply the DVFS scheme to all the benchmark programs and assume there is no frequency specification from the user. Four Pareto fronts are constructed for each program using the four types of explorations: forward and backward searches with WCRT or WCEC as the constraint. The increment step of the constraint is chosen such that the (WCRT, WCEC) space is evenly divided into 32 samples (See Algorithm 2). The results are plotted in Fig. 5.12 and 5.13 with all the dominated points removed.

We then merge all the Pareto optima for each benchmark program, removing all the dominated points, and plot them in Fig. 5.14 for comparing with the two existing approaches.

The fixed frequency approach

The fixed frequency approach serves as a baseline for comparison, which represents the case of executing a conventional synchronous program at a fixed frequency, without using DVFS. There is no frequency switching penalty, and the WCRT and WCEC are computed using the ILP technique presented in Section 5.4. The fixed frequency approach is benchmarked for
each of the four available frequencies for each program. The results are plotted in the (WCRT, WCEC) space in Fig. 5.14 as orange triangles. All these four points are strong Pareto points, meaning that there does not exist any point better in both criteria WCRT and WCEC. In Fig. 5.14, we also outline the portion of the (WCRT, WCEC) plane dominated by each fixed frequency point.

The linearised approach

The linearised approach handles the bicriteria optimization problem by first linearising the program execution path using profiling [65,66]. To implement this principle, we use the critical path obtained through conventional WCRT analysis as the profiled path, and we optimize the energy over that path under a set of WCRT constraints. These constraints are the same as those used in the proposed approach. We also restrict the allocation of DVFS control points to be at the start, EOT and join nodes to be consistent with the proposed approach. The results are plotted in Fig. 5.14 as green crosses.

5.6.2 The Pareto fronts generated by the proposed technique

We apply the proposed bicriteria optimisation technique to all the benchmark programs, each with the four types of exploration. The results are plotted in Fig. 5.12 and Fig 5.13. From these results, we observe that using WCRT as the constraint (Fig. 5.12) generates more evenly distributed schemes across the space compared with using WCEC (Fig. 5.13). This is particularly evidential in the Channel Protocol and Robot Sonar programs, which have an almost continuous selection of schemes. In the other programs, using WCRT constraints tend to have large horizontal gaps that divides the schemes into segments. In contrast, using WCEC constraints tend to have large vertical gaps.

Comparing the forward and backward searching, they both generate about the same amount of non-dominated schemes across the space. The forward searching generates slightly better schemes on left end of the space, which dominates the schemes generated by backward search. Vice versa, the backward search is slightly better on the right end of the space.

All four explorations are able to generate schemes across the (WCRT, WCEC) space with a similar trend. This trend briefly matches the quadratic relationship between power consumption and processor frequency (See formula in Sec. 5.1.3), which is an indication that the estimated Pareto front is close to the actual one.

Overall, all four explorations can provide schemes across the (WCRT, WCEC) space with different WCRT and WCEC tradeoffs, and each has their own strength and weakness in the continuity.
Figure 5.12: The estimated Pareto fronts of the benchmark programs. Turing WCRT into constraints.
Figure 5.13: The estimated Pareto fronts of the benchmark programs. Turing WCEC into constraints.
Figure 5.14: Merging all the estimated Pareto fronts and compared with two existing techniques.
5.6.3 Comparing with existing approaches

The best schemes produced by the proposed technique are obtained by merging the results from the four explorations, and removing all the dominated schemes. The results of this merging are plotted in Fig. 5.14, which are greatly improved in continuity. To evaluate the optimality of these estimated Pareto fronts, we compared the results with the fixed frequency approach and the linearised approach.

The linearised approach is the worst technique with respect to WCRT and WCEC. Most of the schemes produced by the approach overlap with each other, thus only a few points are visible in each graph. All of these points are dominated by the points produced by the other two approaches. More importantly, due to the abstraction (i.e., considers only the profile path and ignores all the others), the linearised approach misses the given WCRT constraint most of the time.

The fixed frequency approach produced exactly four points in the (WCRT, WCEC) space, one for each processor frequency. These points are proven to be Pareto optima [82], which serve as reference points for the trend of the actual Pareto front. Compared with this fixed frequency approach, the proposed technique is disadvantaged due to the frequency switching penalties, which make up 3%-12% of WCRT. However, we produced a lot more non-dominated points in between the four fixed frequency optima. This is a significant advantage in many cases. For instance, let us consider the Channel Protocol program (Fig. 5.14(a)). If the user wants to achieve a WCEC less than 80 µJ, then the three approaches deliver a program with the following WCRT:

- Linearised approach: WCRT = 29.28 µs
- Fixed frequency approach: WCRT = 20.46 µs
- Proposed approach: WCRT = 18.01 µs

In this case, it is clear that the proposed approach provides the best option as it achieves the smallest WCRT.

5.6.4 Analysis time of the proposed technique

<table>
<thead>
<tr>
<th></th>
<th>Channel Protocol</th>
<th>Flasher</th>
<th>Robot Sonar</th>
<th>Cruise Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCRT forward</td>
<td>7.1</td>
<td>7.99</td>
<td>13.59</td>
<td>16.01</td>
</tr>
<tr>
<td>WCRT backward</td>
<td>9.16</td>
<td>14.55</td>
<td>17.64</td>
<td>35.46</td>
</tr>
<tr>
<td>WCEC forward</td>
<td>8.08</td>
<td>13.08</td>
<td>16.02</td>
<td>36.05</td>
</tr>
<tr>
<td>WCEC backward</td>
<td>7.01</td>
<td>7.61</td>
<td>14.9</td>
<td>17.72</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>31.35</strong></td>
<td><strong>43.23</strong></td>
<td><strong>62.15</strong></td>
<td><strong>105.24</strong></td>
</tr>
</tbody>
</table>

Table 5.7: Analysis time for estimating the Pareto fronts of the benchmark programs. (Unit: seconds)

Analysis time is an important quantitative measure for the practicality of the technique. Off-
5.7 Conclusions

The synchronous paradigm is a top choice for designing safety critical reactive systems, thanks to its underlying formal semantics that allows formal verification of the system properties. However, the synchronous paradigm is incapable of utilising the widely available power management technologies which allows trading between processing power and energy consumption. This puts it at disadvantage when designing battery powered systems.

The goal of this chapter was to overcome this limitation by utilising the DVFS to control the tradeoffs between energy consumption and processing speed, giving the user options to choose a tradeoff to suit the specific need. To achieve the goal, we proposed a DVFS scheme that is tightly coupled with the synchronous paradigm. This scheme preserves the predictability of the paradigm which allows both timing and energy consumption to be computed statically. We have also developed an associated bicriteria optimisation technique to construct the Pareto front in a (WCRT, WCEC) space. Through benchmarking, the proposed DVFS scheme and bicriteria optimisation technique have proven their effect. Compared with using a single frequency and the linearised approach [65, 66], the proposed technique provided a lot more options for the user to choose from with various tradeoffs in WCRT and WCEC. In many cases, these options are far better choices. While the search space of the problem is exponential, the proposed technique remains partial to use in real-life, thanks to the employed heuristics. On average, the analysis time is only around one minute for our benchmark program.

To the best of our knowledge, this is the first bicriteria (WCRT, WCEC) optimization method for synchronous programs. This enables the synchronous paradigm to be used in designing a battery powered safety critical system, and potentially, being extended further to optimise for more criteria, such as combining with profiling techniques to optimise for the average energy consumption of the program.
Conclusions

The synchronous paradigm has been a top choice for designing safety critical systems. Its underlying semantics enable formal verification of functionalities and timing, setting the paradigm apart from all other design approaches. However, in today’s competitive environment, it shows two major shortcomings. Firstly, existing Worst-Case Reaction Time (WCRT) analysis techniques cannot handle large modern systems without significantly reducing their precision. Secondly, there is no support of power management in the synchronous paradigm, making it less suitable for battery-powered systems. In this thesis, we addressed these two shortcomings.

In Chapter 3, we presented our first attempt to develop a scalable WCRT analysis technique. The technique is called ILP_C, which is completely based on implicit path enumeration. This is the first WCRT analysis technique that employs an iterative framework to improve the scalability. In ILP_C, we significantly extend the conventional Implicit Path Enumeration Technique (IPET) with a new Integer Linear Programming (ILP) formulation that natively captures the high-level control flow of synchronous programs, resulting in better precision and yet shorter analysis time. We also developed a model called tick expressions to efficiently capture the tick alignment [58, 59] of the program. This allows us to isolate the execution of each node and verify an execution path using only a small set of information. Both the new ILP formulations and tick expressions have jointly contributed toward a scalable WCRT analysis technique.

In Chapter 4, we presented our second attempt to scalable WCRT analysis: WCRT algebra. This time, the technique is based on explicit path enumeration. The most well-known problem with explicit path enumeration is the state explosion problem, which is caused by the combinatorial states of concurrent threads. We proposed the idea of using a timing equivalent automaton,
Tick-Cost Automaton (TCA), for WCRT analysis. TCA is formalised using min-max-plus algebra, and we have developed a technique to systematically transform between TCA and Timed Concurrent Control Flow Graph (TCCFG). Compared with the conventional approach, which uses a functional equivalent automata, TCA considers how the control flow affects the timing instead of functionalities. In theory, TCA captures the exact same information as conventional functional equivalent automata, but due to the more compact mathematics, TCA often requires far fewer operations for the analysis. As a result, we can greatly mitigate the state explosion problem without losing precision in most cases.

Both ILP_C and WCRT algebra are benchmarked against the published state of the art WCRT techniques, and the results show that both techniques are scalable while maintaining the same precision as the existing techniques. For large programs where the existing techniques fail, due to impractical analysis time or insufficient memory, ILP_C and WCRT algebra can complete the analysis very efficiently. The analysis time for those programs are orders of magnitude faster than the existing approaches.

In Chapter 5, we proposed a Dynamic Voltage and Frequency Scaling (DVFS) scheme for the synchronous paradigm and a bicriteria optimisation technique to address the power management problem. We proposed a compile-time DVFS scheme that is closely linked to the synchronous semantics, allowing user to fully or partially control the processor frequency within the synchronous program (i.e., high-level context). We also developed a bicriteria optimisation technique with a greedy heuristic to explore possible settings in the (WCRT, WCEC) space using the concept of Pareto optimality. Comparing with using a single frequency and the existing approach, our technique can provide significantly more options for the user.

To this end, we have explored the potential for both implicit and explicit path enumeration for WCRT analysis, and have successfully developed techniques that are both scalable and precise. Also, we bridged the gap between the synchronous paradigm and power management. With our techniques, the synchronous paradigm can be effectively used for designing battery powered systems. These developed techniques significantly enhances the synchronous design flow, making it more versatile in facing today’s challenges.

Future work

While powerful, the techniques presented in this thesis still have limitations and can be further improved. Firstly, both ILP_C and WCRT algebra can be extended to support the modelling of data-flow in synchronous programs. This can be easily done with ILP_C, since the technique is based on ILP. There exists a body of literature that use ILP to capture the data-flow [87,88,89], and these techniques can be directly used in ILP_C. As for WCRT algebra, it can be extended by adding the formulations similar to [76]. This will require extending the operators and elements in the algebra structure, and combining the algebra formulations.

The second is to apply the knowledge to a multi-core platform. Multi-core is a different research domain. Comparing to the background of this thesis, it will require a new processor,
a different synchronous language that supports multi-core processors, and a different design flow. All these differences imply that the proposed techniques will not work for multi-core processor. However, the proposed techniques achieve their scalability through exploiting the concurrency inherent in synchronous programs. It may be beneficial to reuse some of the principles to establish new research.

Lastly, the proposed techniques support only one synchronous language Precision Timed C (PRET-C). We can extend them to support a wider range of synchronous languages, such as Esterel.
References


REFERENCES


