People interact with a diverse range of embedded systems on a daily basis. Such systems are found in vehicles, robots, and biomedical devices. An embedded system is safety-critical if its failure to operate correctly can lead to dangerous situations that endanger people’s lives or cause financial losses. Safety-critical systems must be certified against strict safety standards before use. Fairly recently, the concept of cyber-physical system (CPS) has emerged that emphasizes the tight integrations between embedded systems and their physical environment. A CPS must deliver its computed outputs on time according to its system specifications. Thus, a key to building a successful CPS is to understand the timing behavior of its computations and physical environment.

The use of high-performance and low-power multi-core processors is becoming popular in embedded systems, triggering the need for embedded programmers to become parallel programming experts. Parallel programming is challenging because programmers must have the requisite skills, experience, and knowledge to avoid common parallel programming traps and pitfalls. Programmers need to have a strong comprehension of parallelism to develop robust parallel programs and not be overwhelmed by its complexity. Programmers need to be aware of the data dependencies in their specific program and choose the appropriate solution to manage the dependencies.

This thesis makes inroads to the future development of CPSs and attempts to unify various research fields by proposing the ForeC language, ForeMC framework, and ForeCast static timing analyzer. The ForeC language enables the deterministic, parallel, and reactive programming of parallel architectures. The synchronous semantics of ForeC is designed to greatly simplify the understanding and debugging of parallel programs. ForeC allows programmers to express many forms of parallel patterns while ensuring that ForeC programs can be compiled efficiently for parallel execution and be amenable to static timing analysis. ForeC’s main innovation is its shared variable semantics that provides thread isolation and deterministic thread communication. All ForeC programs are correct by construction and deadlock-free because mutual exclusion constructs are not needed. Benchmarks on an Intel desktop quad-core processor reveal that ForeC programs can achieve an average speed up of 3.22× compared with 2.83× for the same programs written in OpenMP (a
popular multi-threading framework). On a Xilinx embedded quad-core processor, ForeC programs can achieve an average speed up of $3.49 \times$ compared with $2.25 \times$ for the same programs written in Esterel (a classic synchronous language).

Many CPSs are now comprised of components with various timing requirements, e.g., hard, soft, or non-real-time requirements. The synchronous abstraction, that ForeC is based on, stipulates that programs always react instantaneously to new environmental inputs. This means that all synchronous programs have hard real-time requirements, which precludes the use of soft or non-real-time threads. The ForeMC framework relaxes the synchronous abstraction to allow the definition of threads with hard, soft, or non-real-time requirements. For the first time, the ForeMC framework allows the synchronous language community to venture into the realm of mixed-criticality systems. With size, weight, and power concerns becoming more important, the proposed hybrid (static and dynamic) scheduling approach achieves high processor utilization while maintaining execution fairness among the threads. Simulation-based benchmarks are performed to assess the performance of the hybrid scheduling approach on on multi-processor systems. Benchmarks reveal that the ForeMC framework can schedule up to 15% more task sets and achieve an average of 5.38% better system utilization than a completely dynamic scheduling approach based on earliest deadline first (EDF). Tasks are scheduled fairer under ForeMC and achieve consistently higher execution frequencies, but require more preemptions than an EDF-based approach.

The correctness of a CPS depends on its ability to complete computations in a timely manner. It is therefore critical to analyze the execution times of the programs in a CPS. The ForeCast static timing analyzer verifies the timing requirements of parallel programs by using the reachability technique to compute their worst-case execution times (WCETs). While traversing the program’s control-flow graph, ForeCast simultaneously resolves the execution times of each core due to instruction execution, shared bus delays, synchronization delays, and thread scheduling behavior. We describe the ability to trade-off the analysis precision with analysis time by changing the level of abstraction. Benchmarks on a Xilinx embedded multi-core processor reveal that ForeCast can compute the WCETs of large multi-core programs with an average over-estimation of only 2%. By increasing the level of abstraction, the analysis time for the largest benchmark program (with 43,695 reachable states) is reduced by a factor of 342, to only 7 seconds.

This thesis concludes with a vision for creating an integrated tool chain with ForeC, ForeMC, and ForeCast that helps to simplify the CPS development process.
Acknowledgements

A thesis is a contribution of knowledge to the citizens of this world, built on the contributions of many others. Therefore, this thesis would not have been possible without the support and guidance of Partha S. Roop, my main supervisor, and Morteza Biglari-Abhari, my co-supervisor. I have been lucky that the expertise of both my supervisors have been complementary. Partha, I thank you for thinking continuously about my work, helping me untangle my thoughts onto paper, and walking me through the tough times. Morteza, I thank you for the insightful discussions, words of wisdom, and encouragements.

I am thankful for the financial support that I have received from the University of Auckland Doctoral Scholarship and from the PReSS Account.

I am grateful for the opportunities I have had to collaborate with distinguished researchers from overseas. In particular, Alain Girault from INRIA, Grenoble, France has been pivotal in his contributions to the ForeC semantics and has been a great educator. I must acknowledge the generosity of INRIA and the POP-ART and RIPPES research teams for supporting me with their expertise and funding. David Broman, a visiting scholar at UC Berkeley, USA from Linköping University, Sweden, has been instrumental in advancing my work on mixed-criticality systems. I thank Edward Lee and the CHESS research group at UC Berkeley for enabling this collaboration.

My appreciation goes to the past and present members of the PRETzel research group for the fruitful discussions. I have had the pleasure of collaborating with Matthew Kuo, Sidharta Andalam, Simon Yuan, and Avinash Malik. They have also been invaluable mind readers and proof readers.

I would like to acknowledge my friends who have kept me in touch with reality and for their continuous curiosity in my research. They always ask the hardest question, “When are you finishing?” I have had the privilege of being involved in the Spark entrepreneurship challenge which has helped me balance my work-life.

Finally, the support of my family has been immeasurable. Thanks mum, for always reminding me to eat and sleep!

Eugene Kin Chee Yip


Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in 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.

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.

Chapters 5 and 6.3 of this thesis are based on the following co-authored work:

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation, experiments, and writing the paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>65%</td>
</tr>
</tbody>
</table>

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation, and proof reading the paper.</td>
</tr>
<tr>
<td>Morteza Biglari-Abhari</td>
<td>Initial problem formulation, and proofreading the paper.</td>
</tr>
<tr>
<td>Alain Girault</td>
<td>Problem formulation, and proof reading 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
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partha S. Roop</td>
<td>Click here</td>
<td>26/11/2014</td>
</tr>
<tr>
<td>Morteza Biglari-Abhari</td>
<td>Click here</td>
<td>27/11/2014</td>
</tr>
<tr>
<td>Alain Girault</td>
<td>Click here</td>
<td>26/11/2014</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in 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.

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.

Chapters 7 and 8 of this thesis is based on the following paper:

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Extent of contribution by PhD candidate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem formulation, experiments, and writing the paper.</td>
<td>40%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CO-AUTHORS</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew M. Y. Kuo</td>
<td>Problem formulation, experiments, and proof reading the paper.</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation and proof reading the paper.</td>
</tr>
<tr>
<td>David Broman</td>
<td>Problem formulation and proof reading 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
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew M. Y. Kuo</td>
<td>[Signature]</td>
<td>11/2/2014</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>[Signature]</td>
<td>26/11/2014</td>
</tr>
<tr>
<td>David Broman</td>
<td>[Signature]</td>
<td>28/11/2014</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in 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.

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 is based on the following unpublished paper:

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation, experiments, and writing the paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>60%</td>
</tr>
</tbody>
</table>

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation and proof reading the paper.</td>
</tr>
<tr>
<td>Alain Girault</td>
<td>Problem formulation and proof reading the paper.</td>
</tr>
<tr>
<td>Morteza Biglari-Abhari</td>
<td>Initial problem formulation and proof reading 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
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partha S. Roop</td>
<td>Signature</td>
<td>26/11/2014</td>
</tr>
<tr>
<td>Alain Girault</td>
<td>Signature</td>
<td>26/11/2014</td>
</tr>
<tr>
<td>Morteza Biglari-Abhari</td>
<td>Signature</td>
<td>27/11/2014</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
This form is to accompany the submission of any PhD that contains research reported in 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.

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.

Chapters 3.2, 4, and 6.2 are based on the following unpublished paper:

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Problem formulation, implementation, experiments, and writing the paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>65%</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Mallik</td>
<td>Problem formulation, assistance with the experiments, and writing some of the paper.</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td>Problem formulation and proof reading the paper.</td>
</tr>
<tr>
<td>Alain Girault</td>
<td>Discussions.</td>
</tr>
<tr>
<td>Simon Yuan</td>
<td>Discussions and assistance with the experiments.</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
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Mallik</td>
<td><img src="signature" alt="Signature" /></td>
<td>12/12/2014</td>
</tr>
<tr>
<td>Partha S. Roop</td>
<td><img src="signature" alt="Signature" /></td>
<td>26/11/2014</td>
</tr>
<tr>
<td>Alain Girault</td>
<td><img src="signature" alt="Signature" /></td>
<td>26/11/2014</td>
</tr>
<tr>
<td>Simon Yuan</td>
<td><img src="signature" alt="Signature" /></td>
<td>05/12/2014</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
# Table of Contents

Table of Contents ......................................................... ix

List of Algorithms ........................................................ xiii

List of Figures ............................................................ xvii

List of Tables ............................................................. xix

List of Abbreviations ....................................................... xxi

1. Introduction ................................................................... 1
   1.1 Cyber-Physical Systems .............................................. 2
   1.2 UAV Running Example .............................................. 3
   1.3 Architectures for Average-Case Performance ................. 5
      1.3.1 Single-Core Architectures ................................... 5
      1.3.2 Timing Anomalies .............................................. 7
      1.3.3 Multi-Core Architectures .................................... 7
   1.4 Predictable Parallel Hardware Architectures ............... 9
      1.4.1 A Multi-Core Processor ...................................... 10
      1.4.2 A Single-Core, Multi-Threaded, PRET Processor ..... 11
   1.5 Parallel Programming of Embedded Processors ............ 11
      1.5.1 Synchronous Languages ................................... 13
   1.6 Programming Safety-Critical Embedded Systems .......... 18
   1.7 Static Timing Analysis ............................................. 18
   1.8 Thesis Proposition ................................................ 20
   1.9 Thesis Organization .............................................. 21

2. Literature Review ....................................................... 23
   2.1 Parallel Programming of CPSs ................................... 23
      2.1.1 C-based Programming ...................................... 23
      2.1.2 Synchronous Programming ................................. 26
      2.1.3 Mixed-Criticality Systems ................................. 28
   2.2 Predictable Hardware Architectures ............................ 30
2.2.1 General-Purpose Processors ............................................. 31
2.2.2 Reactive Processors ..................................................... 32
2.3 Static Timing Analysis .................................................... 32
  2.3.1 WCET Analysis ......................................................... 33
  2.3.2 WCRT Analysis ......................................................... 36
2.4 Discussion ................................................................. 38

3. The ForeC Language ......................................................... 41
  3.1 Overview ................................................................. 41
    3.1.1 Local and Global Ticks ............................................. 46
    3.1.2 Fork/Join Parallelism .............................................. 46
    3.1.3 Shared Variables .................................................. 46
    3.1.4 Copying of Shared Variables ..................................... 49
    3.1.5 Resynchronization of Shared Variables ......................... 49
    3.1.6 Hierarchical Preemption ......................................... 51
    3.1.7 Bounded Loops .................................................... 53
  3.2 Design patterns ........................................................ 54
    3.2.1 Point-to-Point ..................................................... 55
    3.2.2 Broadcast .......................................................... 55
    3.2.3 Fork-Join .......................................................... 55
    3.2.4 Map-Reduce ......................................................... 57
    3.2.5 Software Pipeline ................................................ 57
    3.2.6 Early-Termination ................................................ 57
  3.3 Semantics ............................................................... 59
    3.3.1 Assumptions ........................................................ 60
    3.3.2 Notation ............................................................. 61
    3.3.3 Semantic Functions ............................................... 63
    3.3.4 The Structural Operational Semantics ......................... 66
  3.4 Illustrations .......................................................... 72
    3.4.1 Program One ...................................................... 72
    3.4.2 Program Two ...................................................... 75
  3.5 Definitions and Proofs ................................................. 77
  3.6 Comparison with Esterel and Concurrent Revisions ................. 85
  3.7 Discussion ............................................................ 86

4. Compiling ForeC for Parallel Execution ............................... 89
  4.1 Overview ............................................................... 89
  4.2 Static Thread Scheduling ............................................. 90
# TABLE OF CONTENTS

4.3 Structure of the Generated Program .................................. 93
4.4 The `par` Statement .................................................... 94
4.5 The `pause` Statement .................................................. 98
4.6 Shared Variables .......................................................... 98
4.7 The `abort` Statement ................................................... 99
4.8 Global Tick Synchronization ............................................. 100
4.9 Generating Programs for Desktop Execution ......................... 101
4.10 Discussion ............................................................... 102

5. The ForeCast Static Timing Analyzer ..................................... 105
5.1 Overview ................................................................. 105
5.2 Concurrent Control-Flow Graph ......................................... 107
5.3 Shared TDMA Bus .......................................................... 107
5.4 Reachability Technique .................................................. 110
  5.4.1 Fine-Grained Reachability .......................................... 112
  5.4.2 Coarse-Grained Reachability ...................................... 115
5.5 Optimizations ............................................................ 116
  5.5.1 Merging Computation and Condition Nodes ...................... 117
  5.5.2 Merging Edges ....................................................... 117
5.6 Discussion ............................................................... 117

6. ForeC Benchmarking ......................................................... 121
6.1 Benchmark Programs ..................................................... 121
6.2 Performance Evaluation .................................................. 122
  6.2.1 Comparison with Esterel ........................................... 123
  6.2.2 Comparison with OpenMP .......................................... 126
6.3 Discussion ............................................................... 127
6.4 Static Timing Analysis ................................................... 128
  6.4.1 Precision of the Computed WCRT .................................. 129
  6.4.2 ForeCast Analysis Time and Memory Usage ...................... 131
  6.4.3 WCRT Speedup ....................................................... 137
6.5 Discussion ............................................................... 137

7. The ForeMC Framework ..................................................... 139
7.1 Task Model ............................................................... 139
  7.1.1 Extension to More Levels of Criticality .......................... 141
7.2 Delayed Communication Model .......................................... 143
  7.2.1 Multi-Rate Communication Model .................................. 143
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.2</td>
<td>Mixed-Criticality Communication Model</td>
<td>146</td>
</tr>
<tr>
<td>7.3</td>
<td>Scheduling ForeMC for Parallel Execution</td>
<td>148</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Scheduling Approaches</td>
<td>150</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Hyper and Base Period Approaches</td>
<td>151</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Static Scheduling Algorithm</td>
<td>152</td>
</tr>
<tr>
<td>7.3.4</td>
<td>GEN_SCHEDULE</td>
<td>153</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Runtime Scheduling</td>
<td>157</td>
</tr>
<tr>
<td>7.3.6</td>
<td>Heuristics</td>
<td>158</td>
</tr>
<tr>
<td>7.4</td>
<td>Discussion</td>
<td>159</td>
</tr>
<tr>
<td>8.</td>
<td>ForeMC Benchmarking</td>
<td>163</td>
</tr>
<tr>
<td>8.1</td>
<td>Schedulability</td>
<td>165</td>
</tr>
<tr>
<td>8.2</td>
<td>Effects of Varying prop(life)</td>
<td>167</td>
</tr>
<tr>
<td>8.3</td>
<td>Effects of Varying Normalized System Utilization</td>
<td>170</td>
</tr>
<tr>
<td>8.4</td>
<td>Effects of Including Non-Critical Tasks</td>
<td>170</td>
</tr>
<tr>
<td>8.5</td>
<td>ILP Scalability</td>
<td>172</td>
</tr>
<tr>
<td>8.6</td>
<td>Response Times and Preemptions</td>
<td>173</td>
</tr>
<tr>
<td>8.7</td>
<td>Discussion</td>
<td>174</td>
</tr>
<tr>
<td>9.</td>
<td>Conclusions and Future Directions</td>
<td>175</td>
</tr>
<tr>
<td>9.1</td>
<td>Future Directions</td>
<td>177</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Formal Semantics</td>
<td>177</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Thread Scheduling</td>
<td>177</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Static Analysis</td>
<td>179</td>
</tr>
<tr>
<td>9.1.4</td>
<td>Integrated Development Environment for CPS Development</td>
<td>180</td>
</tr>
<tr>
<td>9.2</td>
<td>Final Remarks</td>
<td>181</td>
</tr>
<tr>
<td>A.</td>
<td>Publications</td>
<td>183</td>
</tr>
<tr>
<td>B.</td>
<td>Shared Variables</td>
<td>185</td>
</tr>
<tr>
<td>B.1</td>
<td>Combining More Than Two Copies</td>
<td>185</td>
</tr>
<tr>
<td>B.2</td>
<td>Combine Policies Illustrated</td>
<td>187</td>
</tr>
<tr>
<td>B.3</td>
<td>Examples of Combine Functions</td>
<td>189</td>
</tr>
<tr>
<td>References</td>
<td>193</td>
<td></td>
</tr>
</tbody>
</table>
**List of Algorithms**

1. `GETVAL(E, I, id, var)`: Gets the value of a given variable. ....... 64
2. `COPY(E, t)`: Copies all the shared variables needed by a thread. ... 65
3. `COMBINE(E, t_1, t_2, t_0)`: Combines the copies of shared variables from two threads. ......................... 65
4. Obtains a static schedule for the base period approach. ............. 153
List of Figures

1.1 Examples of embedded systems and those with safety-critical concerns. ......................................................... 1
1.2 Architecture of cyber-physical systems. .............................................. 2
1.3 High-level functional block diagram of the UAV running example. 4
1.4 Speculative features in a single-core architecture. ..................... 6
1.5 Example of a timing anomaly involving the instructions $A$, $B$, $C$, and $D$. Based on [1]. ............................... 8
1.6 General multi-core architecture. .................................................. 8
1.7 Examples of predictable embedded architectures. ....................... 10
1.8 Multi-threaded execution behavior of the PTARM processor. ....... 10
1.9 Programming of parallel architectures. ..................................... 12
1.10 Examples of concurrency errors for a shared variable $x$. ............. 14
1.11 Synchronous model of computation. ........................................... 14
1.12 Examples of synchronous programs. .......................................... 15
1.13 Possible probability distribution of a program’s execution times. .... 19
1.14 Overview of the static timing analysis process. .......................... 19

3.1 Syntactic extensions to C. .......................................................... 42
3.2 Tasks of the UAV. ................................................................. 43
3.3 Example ForeC program for the UAV running example. ............... 44
3.4 Possible execution trace for Figure 3.3. ..................................... 44
3.5 Thread genealogy for Figure 3.3. .............................................. 47
3.6 Figure 3.3 extended with preemption. ....................................... 50
3.7 Possible execution trace for Figure 3.6. ..................................... 52
3.8 Abort variants. ........................................................................ 52
3.9 Nesting of preemptions. ............................................................ 53
3.10 Instances of the design patterns in ForeC pseudo-code. .............. 56
3.11 Example of transforming and translating a ForeC program into the kernel constructs. ........................................... 61
3.12 Retrieving statically known information about Figure 3.11c. ...... 63
3.13 Program one. ................................................................. 72
3.14 Initial program environment and its derivatives. ....................... 73
3.15 Program two. .............................. 75
3.16 Initial program state and its derivatives. ......................... 75
4.1 Overview of compiling ForeC programs. ......................... 89
4.2 Example ForeC program to be compiled. ......................... 91
4.3 Definition of a linked list node and its operations. ............ 93
4.4 Example of the C program generated for Figure 4.2a. .......... 95
4.4 (Continued.) Example of the C program generated for Figure 4.2a. 96
4.5 C code for the immediate and weak variants of the abort on lines 147–162 of Figure 4.4. ........................................ 100
4.6 Using Pthreads to adapt the generated code for desktop multi-cores. 101
4.7 Example Pthreads program. ........................................ 102
5.1 Overview of the ForeCast analyzer. .............................. 105
5.2 Constructing the CCFG of a ForeC program. ..................... 108
5.3 Timing behavior of a shared TDMA bus. .......................... 109
5.4 Using reachability to find ticks in a CCFG. ...................... 111
5.5 Example of computing the WCET of Tick 1b from Figure 5.4e. ... 113
5.6 Reduced CCFG of Figure 5.4a. ................................. 115
5.7 CCFG optimizations. ............................................. 117
6.1 MicroBlaze multi-core configuration. ............................. 123
6.2 Average speedup results for ForeC, Esterel, and OpenMP on four cores normalized to sequential runtime. ..................... 125
6.3 Intel multi-core configuration. ................................. 126
6.4 MicroBlaze multi-core configuration. .......................... 128
6.5 WCRT results for the benchmark programs in clock cycles. .... 130
6.6 Analysis times and memory usage for 802.11a. ................. 132
6.7 Analysis times and memory usage for FlyByWire. .............. 133
6.8 Analysis times and memory usage for Life. ..................... 134
6.9 Analysis times and memory usage for FmRadio. ............... 135
6.10 Analysis times and memory usage for MatrixMultiply. ......... 136
6.11 802.11a’s computed and observed WCRT speedups. .......... 136
7.1 Tasks of the UAV with criticality levels and frequency bounds. . 141
7.2 Example ForeMC program for the UAV running example. ....... 142
7.3 Task communication models. Tasks depicted as executing on their own processors. ............................... 144
7.4 Circular buffer. ............................................. 147
7.5 Static schedules for two life critical tasks on their processor. Note the different time scales. .......................... 151
7.6 An early-release EDF task. ........................................ 151
7.7 Static schedules for the UAV example. .......................... 155
7.8 Runtime traces of two different base periods to show the scheduling decisions that a processor makes. ...................... 157
7.9 Ordering the tasks in the static scheduling phase to improve the dynamic scheduling phase. .......................... 159

8.1 Comparison of the low (ER-EDF) and mission (ForeMC) critical tasks. ......................................................... 164
8.2 Acceptance ratio when the normalized system utilization and prop(life) are varied. .................................................. 166
8.3 System runtime utilization, overall frequency improvement, and fairness when prop(life) is varied. Normalized system utilization = 50%. ......................................................... 168
8.4 System runtime utilization, overall frequency improvement, and fairness when normalized system utilization is varied. prop(life) = 50%. ......................................................... 171
8.5 System runtime utilization, overall frequency improvement, and fairness when prop(non-critical) is varied. Norm. sys. utilization = 50%. ......................................................... 171
8.6 Times for Gurobi to find the first locally and globally optimal solutions. Note the logarithmic time scale. .......................... 173
8.7 Number of preemptions when normalized system utilization is varied. prop(life) = 50% and $N = 4$. .......................... 173

B.1 Example of combining multiple copies of a shared variable. .... 186
B.2 Example of counting the number of button inputs. ............... 188
B.3 Examples of combine functions for primitive C data types. ...... 190
B.4 Examples of combine functions that use pre. .................... 191
B.5 Examples of combine functions for C-structs. .................... 191
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Existing solutions for avoiding race conditions.</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>ForeC constructs and their semantics.</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Existing solutions for avoiding race conditions. (Reproduction of Table 2.1 in Chapter 2.1.1 for the reader’s convenience.)</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Structural translations of bounded loops.</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>ForeC kernel constructs. $f$ is an arbitrary composition of kernel constructs, $var$ is a variable, $exp$ is an expression, $t$ is a thread identifier, and $a$ is an abort identifier. A question mark means that the preceding symbol is optional.</td>
<td>59</td>
</tr>
<tr>
<td>3.5</td>
<td>Structural translations of the ForeC constructs (Table 3.1) to kernel constructs (Table 3.4).</td>
<td>60</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison of ForeC with Esterel and concurrent revisions.</td>
<td>85</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of the synchronization routines.</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>Core 1 and 2’s initial lists and subsequent lists when threads fork.</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of the synchronization routines. (Reproduction of Table 4.1 for the reader’s convenience.)</td>
<td>114</td>
</tr>
<tr>
<td>6.1</td>
<td>Parallel design patterns exploited.</td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>ForeC versus Esterel benchmarks.</td>
<td>124</td>
</tr>
<tr>
<td>6.3</td>
<td>ForeC versus OpenMP benchmarks.</td>
<td>126</td>
</tr>
<tr>
<td>6.4</td>
<td>ForeC benchmarks.</td>
<td>128</td>
</tr>
<tr>
<td>6.5</td>
<td>Number of reachable ticks.</td>
<td>131</td>
</tr>
<tr>
<td>7.1</td>
<td>Extending ForeMC to five criticality levels.</td>
<td>142</td>
</tr>
<tr>
<td>7.2</td>
<td>Mixed-criticality communication model.</td>
<td>146</td>
</tr>
<tr>
<td>7.3</td>
<td>Computed parameters for the life and mission critical tasks of the UAV example. $p_b = 10ms$.</td>
<td>153</td>
</tr>
<tr>
<td>B.1</td>
<td>The pre of count for each combine policy.</td>
<td>188</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
<td></td>
</tr>
<tr>
<td>BCET</td>
<td>best-case execution time</td>
<td></td>
</tr>
<tr>
<td>CCFG</td>
<td>concurrent control-flow graph</td>
<td></td>
</tr>
<tr>
<td>CFG</td>
<td>control-flow graph</td>
<td></td>
</tr>
<tr>
<td>CGR</td>
<td>coarse-grained reachability</td>
<td></td>
</tr>
<tr>
<td>CPS</td>
<td>cyber-physical system</td>
<td></td>
</tr>
<tr>
<td>EDF</td>
<td>earliest deadline first</td>
<td></td>
</tr>
<tr>
<td>ER-EDF</td>
<td>early-release earliest deadline first</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>first-fit (with decreasing criticality)</td>
<td></td>
</tr>
<tr>
<td>FGR</td>
<td>fine-grained reachability</td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>first in, first out</td>
<td></td>
</tr>
<tr>
<td>FSM</td>
<td>finite state machine</td>
<td></td>
</tr>
<tr>
<td>GALS</td>
<td>globally asynchronous, locally asynchronous</td>
<td></td>
</tr>
<tr>
<td>GCD</td>
<td>greatest common divisor</td>
<td></td>
</tr>
<tr>
<td>GRC</td>
<td>GRaph Code</td>
<td></td>
</tr>
<tr>
<td>ILP</td>
<td>integer linear programming</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>instruction set architecture</td>
<td></td>
</tr>
<tr>
<td>LCM</td>
<td>lowest common multiple</td>
<td></td>
</tr>
<tr>
<td>PRET</td>
<td>PREcision Timed</td>
<td></td>
</tr>
<tr>
<td>PROMPT</td>
<td>PRedictability Of Multi-Processor Timing</td>
<td></td>
</tr>
<tr>
<td>PTARM</td>
<td>PRET ARM</td>
<td></td>
</tr>
<tr>
<td>QoS</td>
<td>quality of service</td>
<td></td>
</tr>
<tr>
<td>RTOS</td>
<td>real-time operating system</td>
<td></td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
<td></td>
</tr>
<tr>
<td>SWaP</td>
<td>size, weight, and power</td>
<td></td>
</tr>
<tr>
<td>TDMA</td>
<td>time division multiple access</td>
<td></td>
</tr>
<tr>
<td>UAV</td>
<td>unmanned aerial vehicle</td>
<td></td>
</tr>
<tr>
<td>WCEC</td>
<td>worst-case energy consumption</td>
<td></td>
</tr>
<tr>
<td>WCET</td>
<td>worst-case execution time</td>
<td></td>
</tr>
<tr>
<td>WCRT</td>
<td>worst-case reaction time</td>
<td></td>
</tr>
</tbody>
</table>
People interact with a diverse range of *embedded systems* in their daily lives. These are digital systems that are embedded into a product to add specific functionality. Examples are found in vehicles (e.g., anti-lock braking and electronic fuel injection systems), avionics (e.g., auto-pilot and fly-by-wire systems), phones, cameras, biomedical devices, and unmanned aerial vehicles. The continued miniaturization and cost reduction of digital hardware has allowed embedded systems to be continually optimized for size, weight, and power (SWaP) without increasing costs. An embedded system is *safety-critical* if its failure to operate correctly may lead to catastrophic consequences [2]. Safety-critical systems [3] must be certified against strict safety standards, such as DO-178B [4], IEC 61508 [5], and ISO 26262 [6]. Figure 1.1 gives some examples of embedded systems.

![Embedded Systems](image)

**Figure 1.1**: Examples of embedded systems and those with safety-critical concerns.
1.1 Cyber-Physical Systems

Fairly recently, the concept of cyber-physical systems (CPSs) [7] has emerged that emphasizes the integration of computations (cyber) and physical processes, shown in Figure 1.2. In a CPS, networked embedded systems collaborate to monitor (using sensors) and control (using actuators) physical processes in the environment that in turn affect the computations performed by the embedded systems. Because of the tight coupling between the computations and physical processes, the embedded systems need to be real-time and reactive, computing new outputs as soon as new inputs are detected. A CPS needs to be designed with the awareness that embedded systems evolve in discrete-time because computations take time to complete [8, 9]. On the other hand, the physical environment evolves in continuous-time and the physical processes evolve concurrently, handled by one or more embedded systems. CPSs can be modeled as hybrid systems [10] consisting of continuous and discrete components.

For a CPS, the correctness of its operation depends on the output of its computations and on the timeliness of completing the computations. Consider a cardiac pacemaker [11] with the main functionality of pacing the heart whenever an expected heartbeat is absent. Ideally, the pacemaker should detect an absent heartbeat within 5 milliseconds [12] to allow the heart to be paced at a natural rhythm. If a significant amount of computation time is required to detect an absent heartbeat, then the pacing of the heart can be out of phase with the heart’s beating cycle and cause discomfort for the patient. As a contrasting example, consider an electronic fuel injection system [13] with the main functionality of controlling the amount of fuel injected into each engine cylinder. To control the fuel, it is
necessary to have real-time data about the fuel’s behavior, such as its speed and viscosity along the injection pipes. Because computations take time, new control parameters have to be computed before the next fuel injection phase begins using predicted fuel behavior. The behavior can be predicted using runtime simulation with a constant time step. Since the predicted data is only valid for the end of the time step, the new control parameters only need to be computed by the end of the time step. The system does not benefit by completing the computations earlier.

A key to building a successful CPS is understanding the timing behavior of computations and physical processes. The physical processes, e.g., in a heart or engine, can be described conveniently using ordinary differential equations. Time is an input to these equations, from which the state of the heart or engine can be derived. This makes the timing behavior of physical processes easy to analyze. Unfortunately, the timing behavior of computations modeled in the C programming language [14], a popular language for programming embedded systems, is much more complex to understand. This is because the notion of time is absent (not an input) in C. In essence, the execution time of a C program is nothing more than the side-effect of executing the program on a particular processor [9]. Executing on a different processor may yield a different time, making the modeling of discrete-time dynamics on embedded systems very challenging. This discrepancy in timing behavior is due to various hardware architectural design decisions that favor execution time as a performance metric rather than a correctness criterion. In other words, the design decisions prioritize average-case performance over worst-case performance.

This chapter continues by describing an unmanned aerial vehicle that will serve as the running example throughout this thesis. We describe the general details of hardware designs optimized for average-case performance and the motivation for predictable parallel hardware architectures. We describe the parallel programming of discrete-time models and programming restrictions that ease the certification process. Lastly, we describe the static timing analysis of programs.

### 1.2 UAV Running Example

As the running example for this thesis, we describe the design of an unmanned aerial vehicle (UAV) inspired by the Paparazzi project [15]. A UAV is a remotely controlled aerial vehicle commonly used in surveillance operations. Figure 1.3 illustrates the functionality of the UAV as a block diagram of tasks. The UAV has the following life critical features, where their failure will cause the UAV to
fail. The Navigation task localizes the UAV using on-board sensors, updates the flight path, and sends the desired position to the Stability task. The Stability task controls the flight surfaces to ensure stable flight to the desired position. The UAV has the following mission-critical features, where their failure will cause the UAV to operate at a reduced capacity. The VideoStream task streams a video of the UAV’s flight from an on-board camera to allow users to fly the UAV from the UAV’s point of view. The higher the frame rate, the better the flying experience. The Avoidance task uses on-board sensors to detect obstacles around the UAV and sends collision avoidance data to the Navigation task. The UAV can travel safely at a higher speed when obstacles are checked more frequently. The UAV has the following non-critical features, where their failure has no impact on the UAV’s operation. The Sharing task allows the UAV to share obstacle and localization data with nearby UAVs. Because the UAV combines tasks of different criticalities, it is an excellent example of a mixed-criticality system.

The UAV tasks have a range of timing requirements. The Navigation and Stability tasks control the physical dynamics of the UAV by computing new control parameters using predicted dynamics of the UAV. The dynamics are predicted by solving ordinary differential equations with a constant time step. As a result, the computed control parameters are only valid for the end of the time step. Hence, both tasks have hard real-time deadlines that recur at fixed peri-
ods equal to the time step. On the other hand, the VideoStream and Avoidance tasks can improve their quality of service (QoS) by completing as fast as possible. Completing the tasks earlier reduces the latency of the video stream and allows the collision avoidance data to be fresher or more up-to-date. Hence, both tasks have soft real-time deadlines that recur at fixed periods. The deadlines are soft real-time because any result computed after the deadline becomes less useful, but not completely useless. Lastly, the Logging and Sharing tasks are non-critical and do not need to meet any deadlines. Faithfully modeling the mixed-criticality nature of the timing characteristics of tasks is one of the focuses of this thesis.

1.3 Architectures for Average-Case Performance

To meet the timing requirements of a CPS, a suitable hardware platform is needed to execute the software. General-purpose processors are commonly used but have architectural features that improve average-case performance at the expense of timing predictability. This section highlights their development.

1.3.1 Single-Core Architectures

We begin by looking at the design of single-core processors [16] and highlighting the design decisions that favor average-case performance. The single-cycle processor uses a basic design that executes one instruction in each clock cycle. However, the processor’s clock period is limited to the time needed to execute the most complex instruction, even if the average instruction only needs a fraction of the time. Improving on the single-cycle processor, the multi-cycle processor divides the execution of an instruction into multiple steps or stages. The advantage is that simpler instructions can be executed in fewer stages and that the clock period is only limited by the slowest execution stage. Thus, higher average-case performance can be achieved than the single-cycle processor. Improving on the multi-cycle processor, the pipelined processor overlaps the execution of several instructions whenever possible to keep each stage busy in each clock cycle. The instruction throughput is increased because an instruction can usually start executing in every clock cycle. This is unlike the multi-cycle processor, which waits for the current instruction to complete before executing the next instruction.

Unfortunately, the pipelined design introduces pipeline-related data, control, and structural hazards. A data hazard occurs when an instruction requires data
Figure 1.4: Speculative features in a single-core architecture.

that is still being computed by a later pipeline stage. A control hazard occurs when a branch instruction causes the processor to fetch instructions from a new control-flow, disrupting the smooth flow of the pipeline. Instructions from the old control-flow have to be flushed from the pipeline. Structural hazards occur when multiple pipeline stages need to use the same resource at the same time, such as single-ported memory for data and instruction, or multi-cycle functional units. A simple remedy for pipeline hazards is to stall the affected pipeline stages until the hazard is rectified. However, this degrades performance. A simplified diagram of a pipelined processor is given in Figure 1.4 to highlight the hazards and common design features for minimizing pipeline stalls. Next, we briefly describe some of these design features.

At runtime, the processor can fetch a group of instructions and dynamically reorder their execution, so as to minimize the occurrence of hazards. This is termed as out-of-order execution. However, the reordered instructions can cause new hazards to occur for the instructions that will be fetched in the future. Control hazards can be minimized by dynamically predicting whether a branch will be taken based on the execution history and is performed by additional logic known as the branch prediction unit. The processor can speculatively execute instructions from the predicted control-flow. If the branch prediction is incorrect, then the pipeline has to be flushed and the speculative results have to be ignored. Data hazards can be minimized by forwarding the data computed by the later pipeline stages to the earlier stages. Structural hazards can be minimized by duplicating the highly contended resources. To maximize the use of the duplicated resources, multiple independent instructions can be sent simultaneously into the pipeline for execution whenever possible. Such architectures are termed as superscalar. The compiler can transform or restructure the programmer’s code to generate a sequence of ma-
Machine instructions that maximizes the processor’s utilization. Accessing off-chip memories is expensive because it can take hundreds of processor clock cycles to complete. To bridge this performance gap, fast (on-chip) caches are used to store commonly used data and instructions. Caches can only store a limited amount of data and instructions, requiring existing data or instructions to be replaced whenever the cache is full. Replacement policies, such as least recently used, decide how the cache contents are replaced at runtime.

1.3.2 Timing Anomalies

The optimizations described above for minimizing pipeline stalls and improving average-case performance have unpleasant side-effects on the processor’s runtime behavior. The processor’s behavior depends partly on the decisions made by the optimizations, which depends on the processor’s execution history. This cyclic dependency makes the processor’s behavior difficult to understand and analyze. Moreover, the side-effects that the optimizations can have on each other is problematic, sometimes causing timing anomalies [1]. A timing anomaly occurs when a local worst-case execution time does not lead to the program’s worst-case execution time. Figure 1.5 shows two execution traces that exemplify a timing anomaly with the superscalar execution of instructions A, B, C, and D. Instruction A loads the value that instruction B needs for an arithmetic calculation. Instruction D stores the arithmetic result of instruction C. The arrows indicate the execution dependencies between the instructions. If instruction A is a cache hit (top trace), then all the instructions execute sequentially. If instruction A is a cache miss (bottom trace), then the execution of A becomes longer. This allows instruction C to execute before instruction B, causing instructions B and D to execute in parallel. Thus, the cache miss (local worst-case) leads to a shorter overall execution time.

1.3.3 Multi-Core Architectures

Embedded systems continue to explode in complexity and functionality [17], motivating the use of high-performing, low-power processors. In pursuit of increasing the average-case performance, hardware designs have used multiple processors in the same system (called a multi-processor system) to allow computations to execute in parallel. It is now common to integrate multiple processor cores on the same processor [18, 19], creating a multi-core processor as illustrated by Figure 1.6. To meet the size, weight, and power (SWaP) concerns, the advent of affordable embedded multi-core processors present designers with the opportunity to achieve
better performance and power efficiency than single-core processors [18]. The cores of a multi-core processor may have shared resources, such as caches, memory, and external peripherals via a high-performance shared bus. For the memory hierarchy, the cores may have multiple levels of private and shared caches and employ cache coherence protocols to ensure data consistency.

Multi-core processors are more difficult to analyze compared to their single-core counterparts because of shared resources. Cores interfere with each other whenever they access the same resource in parallel, causing data corruption or undefined hardware states. For example, cores writing to the same display can cause the graphics to be garbled. Interferences can be avoided by forcing the accesses to proceed at mutually exclusive times or by enhancing the resources to support parallel accesses. However, the overuse of mutual exclusion can degrade parallel performance and introduce complex synchronization patterns. Because cores can interfere with each other at any time, analyzing the execution of programs on multi-cores requires even more effort than analyzing single-cores. To ease the analysis effort, embedded systems that require safety certification use simple multi-core processors with predictable architectural features in their designs. The next section describes the design of predictable parallel hardware architectures.
1.4 Predictable Parallel Hardware Architectures

The previous section highlighted the use of speculative features to improve the average-case performance of processors. Unfortunately, this can result in unpredictable execution behavior and is undesirable for CPSs. The PREcision Timed (PRET) machine \cite{20} and PRedictability Of Multi-Processor Timing (PROMPT) \cite{21} design philosophies aim to tackle this issue by advocating the design of predictable hardware architectures, while not sacrificing performance. In particular, the architecture should provide timing isolation between the cores, such that the actions of the cores do not influence each other’s timing behavior. The architecture should be timing compositional, i.e., free of timing anomalies and repeatable timing behavior. The following are examples of unpredictable hardware features with possible predictable alternatives:

**Replace caches with fast software managed memories, called scratchpads** \cite{22}. The selection of data and instructions to be allocated to a scratchpad is determined entirely at compile time. In the static allocation scheme, the contents of the scratchpad cannot be changed at runtime. In the dynamic allocation scheme, the contents of the scratchpad can be changed at runtime by using compile time decisions. The memory address spaces of scratchpads and global memory are mutually exclusive.

**Replace out-of-order execution with better code generation from the compiler** \cite{23}. A processor’s ability to reorder a group of instructions is limited by the size of its instruction buffer. The compiler does not have this limitation because it has access to the entire program and can make better judgments when reordering instructions. However, it may not have the runtime execution information which may affect the performance.

**Deactivate high-performance bus features, such as burst transfers or pipelining, and use fair, time-sharing arbitration policies, such as round-robin or time division multiple access (TDMA)** \cite{24}. The round-robin policy cycles through a static list of cores, granting them access to the bus. If the granted core does not need the bus, then the grant is given to next core on the list. The TDMA policy cycles through a static list of cores, granting them access for a fixed amount of time (a time slot), whether the core needs it or not. If the granted core does not need the bus, then some policies \cite{24} will grant the slot to the other cores in a round-robin manner,
an arbitrary number of cores, and shared bus with TDMA arbitration. Existing MicroBlaze simulator for benchmarking purposes. We extended an experiment with peripherals. Due to the resource constraints of an FPGA device, we developed a core MicroBlaze simulator using identical Xilinx MicroBlaze. The first predictable processor is a homogeneous multi-core processor that we have designed using identical Xilinx MicroBlaze cores, illustrated by Figure 1.7a. Each MicroBlaze core has a three-stage, in-order, timing anomaly-free pipeline connected to private data and instruction scratchpads. The scratchpads are statically allocated and loaded at compile time. A shared bus with TDMA arbitration connects the cores to shared resources, such as global memory and peripherals. Due to the resource constraints of an FPGA device, we developed a multi-core MicroBlaze simulator for benchmarking purposes. We extended an existing MicroBlaze simulator significantly to support cycle-accurate simulation, an arbitrary number of cores, and shared bus with TDMA arbitration.

Figure 1.7: Examples of predictable embedded architectures.

Figure 1.8: Multi-threaded execution behavior of the PTARM processor.

thus improving the throughput. Fairness of the arbitration is important to ensure that all accesses complete within a bounded length of time.

Embedded systems designed using the PRET or PROMPT philosophies are simpler to understand, model, and analyze. The following two subsections describe two predictable embedded processors that are used in this thesis.

1.4.1 A Multi-Core Processor

The first predictable processor is a homogeneous multi-core processor that we have designed using identical Xilinx MicroBlaze cores, illustrated by Figure 1.7a. Each MicroBlaze core has a three-stage, in-order, timing anomaly-free pipeline connected to private data and instruction scratchpads. The scratchpads are statically allocated and loaded at compile time. A shared bus with TDMA arbitration connects the cores to shared resources, such as global memory and peripherals. Due to the resource constraints of an FPGA device, we developed a multi-core MicroBlaze simulator for benchmarking purposes. We extended an existing MicroBlaze simulator significantly to support cycle-accurate simulation, an arbitrary number of cores, and shared bus with TDMA arbitration.
For all benchmark programs in this thesis, the instructions are placed in each core’s private instruction scratchpad. The stack, heap, and read-only data are placed in each core’s private data scratchpad. Shared and global variables are placed in global memory. We did not encounter any benchmark programs that could not fit their instructions or read-only data in each core’s scratchpads. However, if the instructions or data cannot be placed entirely into a core’s private scratchpads, then the excess instructions or data must be placed in global memory. At runtime, data and instructions must be transferred between the scratchpads and global memory. Dynamic scratchpad allocation algorithms [28, 29, 30] can be used to automatically insert memory transfer instructions [22] into the program code. This can result in longer compilation time, WCRT analysis time, and program execution time.

1.4.2 A Single-Core, Multi-Threaded, PRET Processor

The second predictable processor is a single-core multi-threaded processor based on the PRET architecture, illustrated by Figure 1.7b. We use the PRET ARM (PTARM) processor developed and implemented by Liu et al. [31]. PTARM is a single-core processor with a five-stage, in-order pipeline and implements the ARMv4 instruction set architecture (ISA). PTARM supports the execution of four different hardware threads with timing isolation guaranteed between the threads. Figure 1.8 illustrates the execution behavior of the hardware threads ($t_1$ to $t_4$). Each hardware thread executes in a multi-cycle fashion to prevent pipeline hazards from occurring. The hardware threads are pipelined together to keep the pipeline stages busy and achieve high processor utilization. All the hardware threads share a statically allocated scratchpad for data and instructions. A feature of PRET processors is the inclusion of timing instructions in the ISA that facilitates the modeling of time-dependent behaviors.

1.5 Parallel Programming of Embedded Processors

This section introduces some methods for creating programs that can take advantage of the parallel architectures described in the previous sections. Figure 1.9 shows some layers of abstraction that can be used to create such programs. Programs can be executed directly by the hardware (bare-metal) or by a real-time operating system (RTOS) [32]. The main benefit of the bare-metal approach is
that all the system resources can be devoted to executing the program. Programs must, however, include code for managing the hardware. The main benefit with an RTOS is that it manages the hardware and provides a consistent environment for developing and executing programs, thus, enabling code portability across a range of systems. An RTOS also provides a set of convenient functions, such as task scheduling and management. A portion of the system’s resources, however, has to be devoted to running the RTOS. When analyzing programs, the RTOS must be taken into account. C is a popular language for programming embedded systems, but the early C standards [33] do not support multi-threading. Thus, C was extended to support multi-threading and parallelism through third party libraries, compilers, and runtime support from operating systems [34]. Notable examples include Pthreads [35], OpenMP [36], and MPI [37]. The latest C standard, revised in 2011 [14], is the first to introduce multi-threading, but as an optional feature. These multi-threading solutions are inherently non-deterministic [38] because their execution behavior depends on the runtime environment and its implementation. The lack of formal semantics for the programming model can also lead to ambiguous behaviors.

Threads accessing the same shared resource concurrently will interfere and corrupt the resource (also called a race-condition), unless the resource has been designed to handle parallel accesses. A simple example of a race-condition is shown in Figure 1.10a with two threads racing to write a value to x, a shared variable. Depending on how the hardware behaves, the resulting value of x could be 1, or 2, or undefined. It is the programmer’s responsibility to identify the regions of code that can interfere, called critical sections, and ensure that they are executed sequentially at mutually exclusive times. Forcing critical sections to
execute sequentially can lower the program’s performance. The common approach is to use the mutual exclusion constructs provided by the libraries to create locks for accessing each shared resource. A thread must obtain the required lock before it can enter a critical section. The thread releases the lock when it exits the critical section. Unfortunately, this approach is error prone and complex when many threads need to access a range of shared resources. Critical sections that are improperly managed can lead to a range of concurrency errors, such as those shown in Figure 1.10. Figure 1.10b shows atomic violation, where multiple accesses to a shared resource have been incorrectly split into separate critical sections, leading to incorrect program behavior. Figure 1.10c shows order violation, where the program behavior depends on the execution order of the critical sections. Other concurrency errors include deadlock and priority inversion. Deadlock can occur when threads, that need to obtain multiple locks to enter their critical section, obtain each other’s locks and prevent each other from progressing. Priority inversion can occur when a higher priority thread has to wait for a lower priority thread to release a lock. If the higher priority thread is always scheduled before the lower priority thread, then the lock may never be released, causing a deadlock. The need to manage critical sections makes the understanding, debugging, and analysis of parallel programs complex and time consuming [39]. Moreover, having many critical sections in a program can lead to significant synchronization overheads. Studies have shown that, without careful tuning [40], parallel programs executed on multi-cores can perform worse than their sequential counterparts.

Parallel programming is challenging because it requires programmers to have the requisite skills, experience, and knowledge to avoid common parallel programming traps and pitfalls. Programmers need to have a strong comprehension of parallelism to develop robust parallel programs and not be overwhelmed by its complexity. Programmers need to be aware of the data dependencies in their specific program and choose the appropriate solution to manage the dependencies. Programmers need to have access to tools that can test and debug their parallel programs effectively, giving clear diagnostic messages that reveal the cause of each error. The next section describes the use of synchronous languages as an alternative to creating concurrent programs that are deterministic.

### 1.5.1 Synchronous Languages

Synchronous languages [41] are an alternative for creating deterministic and concurrent programs. Synchronous languages are based on sound mathematical semantics, which facilitates system verification by formal methods [41] and the gen-
Figure 1.10: Examples of concurrency errors for a shared variable \( x \).

(a) Race-condition.

(b) Atomic violation for two accesses in Thread 2.

(c) Order violation between two critical sections.

Figure 1.11: Synchronous model of computation.
A thread in the program sample the environment, perform their computations, and emit their results to the environment. When a thread completes its computation, we say that the thread has completed its local tick. When all threads in the program have completed their local ticks, we say that the program has completed its global tick. Central to synchronous languages is the synchrony hypothesis\(^{[41]}\), which states that the execution of each reaction is considered to be atomic and instantaneous. This separates the physical time of the environment from the logical time of the program, thus simplifying the language semantics and enabling formal verification. The progression of time in discrete steps makes synchronous languages suitable for programming discrete-time models. The sampling of inputs avoids the need to use interrupts which are sources of unpredictable delays that degrade the system’s timing predictability. Concurrent threads communicate instantaneously with each other (dashed arrows in Figure 1.11) during their computation due to the synchrony hypothesis. Once the embedded system is implemented, the synchrony hypothesis has to be verified. That is, the worst-case execution time \(^{[44]}\) of any global tick must not exceed the minimal inter-arrival time of the inputs.

We use the Esterel synchronous language \(^{[45]}\) to illustrate some features of
the synchronous paradigm in Figure 1.12a. It contains two threads (starting from lines 4 and 7 respectively), scoped between the square brackets and separated by the parallel operator || (line 6). The parallel operator is commutative and associative and specifies that both threads are executed concurrently. The execution of a thread can be divided over multiple global ticks with the `pause` statement (e.g., lines 5, 10, and 13). The `pause` statement pauses the execution of its enclosing thread, demarcating the end of the thread’s local tick. All executing threads must pause or terminate to complete the global tick. Thus, the `pause` acts as a synchronization barrier. At the next global tick, the threads resume from their respective `pauses`. In Esterel, threads communicate by emitting signals and threads can test for their presence or absence. For example, line 2 declares two signals, A and B, that are emitted by the `emit` statement when execution reaches lines 4, 8, 12, 14, and 15. An emitted signal lasts until the global tick ends, becoming absent in the following global tick unless it is emitted again. Note that emitted signals are present from the start of the global tick to ensure that all the threads see the same signal statuses, even if the `emit` statements occur later in the global tick. In the first global tick, the first thread emits the signal A. At the same time, the second thread tests positively for the presence of A and emits B. Using the `abort` statement, a body of code can be preempted by the presence of a signal. Preemption provides a convenient way to model the transitions and states of a state machine.

In the second global tick of the example program, the second thread enters an `abort` (line 11) if A is present. Because A is not present, the body is not preempted and B is emitted. Meanwhile, the first thread terminates because it has reached the end of its body.

Synchronous programs are notoriously difficult to parallelize [46, 47, 48] due to the need to resolve instantaneous thread communication and associated causality issues. At runtime, all potential signal emitters must be executed before all testers of a signal. If this is not possible, then a causality issue arises. Thus, concurrency is typically compiled away to produce only sequential code [42]. The common approach for parallelizing synchronous programs is to automatically parallelize an intermediate representation of the program [46, 47, 48, 49]. The techniques differ in the heuristics used to partition the program to achieve sufficient parallelism.

To reduce the learning barrier of synchronous languages and to appeal to C programmers, C-based synchronous languages have been proposed. These languages extend C with a range of synchronous constructs to support concurrency, preemption, and thread communication. PRET-C [50] is one such example and Figure 1.12b is the PRET-C version of the Esterel example (Figure 1.12a). The
two threads (t0 and t1, defined on lines 5 and 9) are arguments to the parallel operator PAR (line 3). The EOT statement demarcates the end of a thread’s local tick (e.g., lines 7, 13, and 16). Unlike Esterel, threads in the PAR’s argument are executed in a left-to-right (static) order. The thread’s local tick must be executed entirely before the next thread can be executed. In the example program, t0 always executes its local tick before t1. In PRET-C, threads communicate using globally declared C-variables, not signals. Because threads are always executed in a static order, the local ticks always execute in a mutually exclusive manner. Hence, threads can safely access shared variables without needing to use mutual exclusion constructs; all PRET-C programs thread-safe by construction. In the first global tick of Figure 1.12b, t0 executes first and assigns 1 to the shared variable A (line 6). Then, t1 executes and checks the condition A==1, which is true, and assigns 1 to B. PRET-C does not suffer from causality issues because global variables are always present and variables are always accessed sequentially (within a thread and across threads). PRET-C supports preemption with the abort statement but its behavior differs from Esterel’s abort. Preemption occurs when the associated C-condition evaluates to true. The condition is always checked before the abort body is executed. In the second global tick of the example program, t0 terminates because it has reached the end of its body. Then, t1 executes and the condition A==1 (line 19) is checked before the abort body (lines 15–18) is executed. The condition is true, so the body is preempted. Execution jumps to line 18 and t1 terminates.

Other C-based synchronous languages exist, such as Esterel C Language [51] and Synchronous C [52], and are reviewed in Chapter 2.1.2. However, these languages are not designed to take advantage of parallel execution. This thesis focuses on developing a C-based, synchronous language for writing parallel programs that perform well and are amenable to static timing analysis. Moreover, we are interested in relaxing the synchrony hypothesis to enable the specification of hard, soft, and non-real time threads. Under the traditional synchronous framework, all the threads must be hard real-time to satisfy the synchrony hypothesis. Thus, if the UAV example was to be modeled in a synchronous language, then all the threads would have to be hard real-time, leading to the over-provisioning of resources. This thesis presents an approach for preserving the timing requirements (hard, soft, and non-real-time) of threads, so as to minimize the over-provisioning of resources.
1.6 Programming Safety-Critical Embedded Systems

Safety-critical embedded systems need to be certified against stringent safety standards, such as DO-178B [4] or IEC 61508 [5], before they can be deployed and used in the field. Although the C language is popular for programming safety-critical embedded systems, its semantics [14] include unspecified and undefined behaviors [53]. Strict coding guidelines [54, 55, 56] are typically used by safety-critical programmers to help write well-defined programs that are deterministic, understandable, maintainable, and easier to debug [57, 58]. The coding guidelines can be grouped into three main areas:

**Code clarity:** These guidelines suggest a style for writing programs free of ambiguous statements and to structure code for readability. For example, the use of braces to clarify the nesting of if–else statements. Code clarity helps static analyzers parse the program and attain greater analysis precision.

**Defensive programming:** These guidelines help minimize the use of unspecified and undefined behaviors, which contribute towards non-determinism. For example, expressions with side-effects should not be used in function arguments. This is because the evaluation order of function arguments is unspecified in the semantics of C.

**Runtime reliability:** These guidelines help minimize runtime errors from occurring, even when the program is written correctly. For example, a runtime error occurs when a program requests for more memory than is available in the implemented system.

1.7 Static Timing Analysis

The timing behavior of embedded systems in CPSs is of utmost importance due to the tight coupling of computations and physical processes. This was illustrated by the examples in Chapters 1.1 and 1.2. The timing behavior of an embedded system is a function of its program and the processor. Implementation choices, such as those discussed in Chapters 1.3, 1.4, and 1.5, have a large impact on the timing behavior. The execution time of a system can vary greatly depending on the program inputs received at runtime and the resulting processor states. Figure 1.13 exemplifies a program’s execution times as a probability distribution. The solid curve represents the probability distribution of the program’s actual execution
1.7 Static Timing Analysis

Figure 1.13: Possible probability distribution of a program’s execution times.

Figure 1.14: Overview of the static timing analysis process.

times, which considers every possible execution path through the program. The shortest execution time is called the best-case execution time (BCET) while the longest is called the worst-case execution time (WCET). The role of timing analysis is to estimate the BCET and WCET as close to the actual BCET and WCET as possible. A straightforward method is to measure the time it takes for the program to execute under various input sequences. Unless all possible input sequences are used, the measured times are only a subset of the actual execution times, shown by the dotted curve in Figure 1.13. In the case of verifying the synchrony hypothesis, relying on the measured WCET is unsafe because the program could execute for a longer time. When safe timing guarantees are required, calculating the WCET statically (known as static timing analysis [44]) is the mandated approach.

Figure 1.14 shows an overview of the static timing analysis process. The first step is to disassemble the program’s executable binary into the assembly instructions that the processor executes. The assembly instructions may include branches that cause execution to jump around the program. The program’s control-flow must be analyzed to reveal the possible execution paths through the program. Thus, the second step is to analyze the program’s control-flow and reconstruct its control-flow graph (CFG). Annotations can be inserted into the source code to help the CFG reconstruction process. The third and final step is to compute the execution time of all the paths through the CFG using a timing model of the processor,
created from micro-architectural analysis. The timing model of the processor calculates the execution time of an instruction, taking into account dependencies on the processor’s execution history. The longest execution time that is computed is the program’s WCET. The WCET can be computed exactly or approximately. An exact computation requires a detailed reconstruction of the CFG, detailed analysis of the CFG, and cycle-accurate timing model of the processor. Unfortunately, analyzing in such high detail leads quickly to the state-space explosion problem, increasing the time and memory needed to perform the analysis. On the other hand, sound abstractions [59] can be used to create simpler approximations of the CFG and timing model of the processor. The simpler models help reduce the state-space, decreasing the time and memory needed to perform the analysis, but at the expense of possibly less precise WCET calculations.

Static timing analysis has been studied extensively [44] for single-core processors while studies for multi-core processors are gaining prominence. This thesis makes important contributions to the static timing analysis of parallel programs on multi-cores. Compared to single-cores, multi-core execution is more complex to analyze because of 1) inter-core interferences, 2) thread-level parallelism, 3) inter-core synchronizations, and 4) thread scheduling across cores. These issues are reviewed in detail in Chapter 2.3. Hence, the techniques for analyzing single-cores cannot be applied directly to multi-cores. This thesis also focuses on the bare-metal implementation of synchronous programs on predictable processors because this combination enables tight WCET estimations. The task of calculating the WCET of all the possible global ticks in a synchronous program is called worst-case reaction time (WCRT) analysis [60].

1.8 Thesis Proposition

The proposed ForeC programming language, ForeMC framework, and ForeCast static timing analyzer will simplify the parallel programming of CPSs on multi-cores. There are three main contributions of this thesis:

**The ForeC language:** A C-based synchronous language for the deterministic parallel programming of embedded multi-cores. Execution platforms have evolved from single-cores to multi-cores. Hence, all the synchronous languages designed for the single-core era must be reinvented to address the multi-core challenges. To this end, ForeC is designed specifically for the programming of multi-cores. ForeC brings together the formal deterministic semantics of synchronous languages and the benefits of C’s control and data
structures. A key innovation is ForeC’s shared variable semantics that provides thread isolation and deterministic thread communication. Moreover, many forms of parallel design patterns can be expressed in ForeC. We prove that ForeC programs are reactive and deterministic by construction. ForeC can be compiled for direct execution on embedded multi-cores or for execution by an OS on desktop multi-cores. Through benchmarking, we demonstrate that ForeC can achieve better parallel performance than Esterel and OpenMP, while also being amenable to static timing analysis.

The ForeMC framework: A relaxation of the synchrony hypothesis to allow synchronous languages to model threads that have hard, soft, or non-real-time requirements. The synchrony hypothesis requires all threads to be hard real-time, thus, limiting their suitability for programming mixed-criticality CPSs. The proposed relaxation of the synchrony hypothesis overcomes this limitation, enabling the faithful modeling of threads with mixed timing requirements. A lossless communication model is proposed that allows hard and soft real-time threads to communicate deterministically, even when their execution frequencies vary at runtime. With SWaP concerns becoming more important, we propose a hybrid (static and dynamic) scheduling approach that exploits the different timing requirements of threads so as to maximize system utilization.

The ForeCast static timing analyzer: A WCRT analyzer for ForeC programs on time-predictable parallel architectures. Our reachability-based timing analysis is more capable and efficient than the only other known approach [47] for synchronous programs on multi-cores. ForeCast computes precise WCRT estimates by considering the effects of 1) inter-core interference, 2) thread-level parallelism, 3) software-based synchronization, and 4) thread scheduling. Benchmarking reveals that the proposed approach computes very tight WCRTs in a fast and scalable manner, even when the same program is distributed over additional cores. This demonstrates the efficacy of the proposed approach for the design of CPSs using parallel architectures.

1.9 Thesis Organization

This thesis is organized as follows. Chapter 2 provides a detailed literature review of predictable architectures, parallel and synchronous programming languages, mixed-criticality task scheduling, and static timing analysis of parallel programs.
Chapter 3 introduces the ForeC language and uses the UAV running example to highlight key features of ForeC. The formal semantics of ForeC is given, along with proofs for determinism and reactivity. Chapter 4 describes our compilation approach for generating code with good parallel performance and amenable to static timing analysis. Chapter 5 describes the ForeCast timing analyzer and its reachability-based technique. Chapter 6 presents benchmarking results for ForeC’s performance on embedded multi-cores and for our ForeCast analyzer.

Chapter 7 introduces the ForeMC framework and expands on the UAV running example to highlight the ability to model threads with different timing requirements. A multi-processor scheduling approach is proposed to preserve the timing requirements of the threads while maximizing system utilization. Chapter 8 presents benchmarking results for system utilization and scheduling fairness.

Finally, Chapter 9 concludes the thesis along with a discussion on future works.
Cyber-physical systems (CPSs) have recently exploded in complexity and functionality, motivating the use of high-performance and low-power multi-core processors [18, 19]. This has triggered the need for embedded programmers to become parallel programming experts, merging the two previously separate engineering disciplines. Safety-critical embedded systems [3] must be dependable and functionally safe [61, 62, 63] and certified against safety standards, such as DO-178B [4], IEC 61508 [5], and ISO 26262 [6]. The priority inversion error [64] of the NASA Mars Pathfinder [65] is a prominent example of a mission-critical failure. The cardiac pacemaker [11, 66, 67] is a prime example of a safety-critical system, where failure may lead to death [2]. Certification demonstrates that all potential risks and hazards have been assessed and considered in the system’s design and implementation, with safety features included to manage and mitigate runtime failures. Certification is a costly and time-consuming exercise and is exacerbated by the use of multi-core designs. The model-driven engineering approach [68, 69] is gaining popularity and facilitates the development and subsequent certification processes. This chapter reviews three key facets of developing CPSs. First, we review the parallel programming of CPSs using C and various synchronous languages and their associated deployment on parallel architectures. Second, we review several predictable parallel architectures. Third, we review the static timing analysis of parallel programs on parallel architectures, necessary for certification purposes.

2.1 Parallel Programming of CPSs

2.1.1 C-based Programming

The C language [14] is popular among embedded programmers, but its semantics is defined informally in English. In addition, the semantics contains many unknown
and unspecified behaviors that are left for compilers to define. This can lead
to semantic ambiguities and cause subtle or surprising programming pitfalls [53].
When programming safety-critical systems, it is important to avoid the unknown
or unspecified behaviors by following safety-critical coding standards and guide-
lines [57, 58], such as MISRA-C [54], the Power of 10 [55], and the Jet Propulsion
Laboratory (JPL) coding standard [56]. Static analyzers, such as Polyspace [70]
and Parasoft [71], can be used to validate the adherence to a particular coding
guide or to discover potentially unsafe program behaviors. See for example the
work on pointer analysis [72, 73]. Efforts have been made to formalize a subset
of the C language to eliminate semantic ambiguities. Examples include Clight [74],
Cholera [75], and Cyclone [76]. Formal semantics is useful for understanding pro-
gram behavior and verifying the correctness of supporting tools, such as compilers
and static analyzers. Clight, in particular, has been verified by the Coq [77] proof
assistant and has been integrated into the CompCert [78] verified compiler.

The growing use of embedded multi-cores is requiring more programmers to be-
come parallel programming experts. Parallel programming is notoriously difficult
because program behavior is affected by the nuances of accessing shared resources
and thread scheduling. This leads to parallel and concurrency errors and many can
be subtle to debug or non-trivial to resolve [39, 79]. Improperly managed parallel
accesses to a shared variable can cause race conditions and leave the program in
an inconsistent state. Table 2.1 highlights different approaches for enforcing mu-
tual exclusion on shared variables, usually by interleaving the parallel accesses to
enforce a sequence of accesses to the critical sections. The theoretical speedup of
a parallel program executed on multi-cores or multi-processors can be calculated
using Amdahl’s Law [80, 81]. Achieving the theoretical speedup can be difficult
in practice and can sometimes result in slower performance [40]. The latest 2011
revision [14] of the C standard includes support for multi-threading, but again has
no formal semantics. The earlier C standards [33] did not support multi-threading
and third-party libraries, compilers, and runtime supports were needed to write
and deploy multi-threaded parallel programs [34]. Well known examples include:
POSIX Threads (Pthreads) [35], Open Multi-Processing (OpenMP) [36], Message
Passing Interface (MPI) [37], Open Computing Language (OpenCL) [82], Intel
Thread Building Blocks (TBB) [83], and Intel Cilk Plus [84]. Unfortunately, these
third-party solutions lack formal semantics, rendering them largely unsuitable for
safety-critical systems. The runtime supports can incur high overhead costs [85]
but light-weight methods [86, 87, 88, 89] for forking and joining threads is available.
However, these focus on maximizing average-case performance.
Programming Constructs: These are constructs written in the host language to provide mechanisms for the programmer to achieve mutual exclusion. Examples include: locks, monitors, transactional memory, message passing, and parallel data structures. Using these constructs correctly can be tedious and error prone for large programs and may lead to other errors [38, 39, 79], e.g., deadlocks, starvation, or priority inversion.

Language Semantics: The language semantics can have a memory model that defines how threads interact through memory, what value a read can return, and when the value of a write becomes visible to other threads. Although the memory model can prevent race conditions, it may only be suitable for a few types of applications. Examples include: synchronous languages [41], PRET-C [50], Synchronous C [52], SharC [90], Deterministic Parallel Java [91], SHIM [92], ΣC [93], concurrent revisions [94], and Reactive Shared Variables [95].

Static Analysis: A compiler or static analyzer can identify and alert the programmer regarding the race conditions in their program (e.g., Parallel Lint [96]) and may try to resolve them by serializing the parallel accesses for the programmer (e.g., Sequentially Constructive Concurrency [97]). However, programmer guidance is needed for race conditions that cannot be resolved.

Runtime Support: Programs are can be executed on a runtime layer that dynamically enforces deterministic execution and memory accesses. Examples include: dOS [98], Grace [99], Kendo [100], CoreDet [101], Dthreads [102], DOMP [103], and DetMP [104]. However, understanding the program’s behavior at compile time remains difficult because the determinism is only enforced at runtime.

Hardware Support: Parallel accesses can be automatically detected and resolved by the hardware, preventing race conditions from happening. Examples include: Ultracomputer’s combine hardware [105] and certain shared bus arbitration (e.g., round-robin, TDMA, and priority). However, the timing of the parallel accesses affects how they are interleaved.

Table 2.1: Existing solutions for avoiding race conditions.
As argued by Lee [38], the adoption of parallelism in sequential languages, like C, discards important properties, such as determinism, predictability, and understandability. Thus, programmers spend large amounts of time taming the non-determinism in their parallel programs [79]. To help alleviate this issue, runtime environments that enforce deterministic thread scheduling and memory accesses can be used. Such runtime environments have been developed for Linux processes (Deterministic Process Groups (DPG) [98]), Pthreads (Grace [99], Kendo [100], CoreDet [101], and Dthreads [102]), OpenMP (Deterministic OpenMP (DOMP) [103]), and MPI (DetMP [104]). For DPG, Kendo, CoreDet, and Dthreads, all thread interactions are mapped deterministically onto a logical timeline (which progresses independently of physical time). Program execution is divided into alternating parallel and serial phases, similar to the Bulk Synchronous Parallel (BSP) [106] programming model. In the parallel phase, threads execute in parallel until they all reach one of the following synchronization points: a lock, memory access, or statically defined number of executed instructions. Then, in the serial phase, threads take turns to resolve their memory accesses or lock acquisitions. Threads in CoreDet and Dthreads also maintain their own version of the shared memory state, which is resynchronized in every serial phase. This concept is used and formally defined in concurrent revisions [94]. DOMP and Grace differ in that the resynchronization only occurs when threads reach a synchronization construct. However, understanding the program’s behavior at compile time remains difficult because the determinism is only enforced at runtime. Thus, if the program is modified, e.g., to rectify a bug, then a vastly different runtime behavior is possible.

An alternative to library-based support is to extend and modify the C language to support parallelism, such as Unified Parallel C (UPC) [107], ParC [108], Concurrent C [109], and Fork95 [110]. C-based languages with deterministic parallelism have been designed for desktop programming (SharC [111] and Cat [112]) and embedded programming (Software/Hardware Integration Medium (SHIM) [113], Sigma-C (ΣC) [93], and ForkLight [114]). These solutions allow the asynchronous forking and synchronized joining of threads, but lack a convenient mechanism for preempting groups of threads. Unfortunately, the timing predictability of these solutions has not been demonstrated, which is required for programming CPSs.

### 2.1.2 Synchronous Programming

Synchronous languages [41] are grounded in a mathematical framework that facilitates formal verification. They are well suited to the modeling of control-dominated systems [115] and safety-critical systems [41]. Synchronous languages provide de-
terministic concurrency with threads executing synchronously relative to a logical clock. The classic synchronous languages are Esterel [45], Lustre [116], and Signal [117]. *Multi-rate* synchronous languages allow tasks to execute at different logical clock frequencies. Examples include, multi-clock Esterel [118], Lustre, Signal, SCADE [119], Lucy-n [120], Synchronous Data Flow (SDF) [121], and Prelude [122]. Closely related multi-rate languages include Giotto [123], Simulink [124], and OpenStream [125]. For programming distributed systems, the globally asynchronous, locally synchronous (GALS) model of computation allows an entire group of synchronous threads to be executed asynchronously with other groups. Languages based on GALS include FunLoft [126] and SystemJ [127].

However, synchronous languages are domain-specific and bear little resemblance to the C language, impacting their uptake with embedded programmers. To bridge this gap, C-based synchronous languages have been developed, such as Reactive Shared Variables [95], Esterel C Language (ECL) [51], PRET-C [50] and Synchronous C (SC) [52, 97]. The inherent sequential execution semantics of SC, Reactive Shared Variables, and PRET-C renders them unsuitable for multi-core execution. Moreover, concurrency in synchronous languages is a logical concept to help the programmer handle concurrent inputs, rather than a specification for parallel execution. Thus, compilers typically generate only sequential code [42, 118], although some generate concurrent tasks [128, 129, 122, 43] for execution on single-cores. Yuan et al. [48, 130] offer a static and dynamic scheduling approach for Esterel on multi-cores. For the static approach, threads are statically load-balanced across the cores and signal statuses are resolved at runtime. For the dynamic approach, threads that need to be scheduled for execution are inserted into a custom hardware queue accessible to all cores. The dynamic approach has been shown to provide better average-case performance compared to the static approach [130].

The common approach for parallelizing synchronous programs is to parallelize an intermediate representation of the sequentialized code [46, 47, 48, 49, 131, 132, 133, 134]. Multi-threaded OpenMP programs can be generated from the Synchronous Guarded Actions intermediate format [49]. The techniques differ in the heuristics used to partition and distribute the program to achieve sufficient parallelism. The Synchronized Distributed Executive (SynDEx) [135] approach considers the cost of communication when distributing code to each processing element. When distributing a synchronous program, some desynchronization [136, 137, 138] is needed among the concurrent threads. That is, the concurrent threads execute at their own pace, but sufficient inter-thread communication is used to preserve the original synchronous semantics. The use of *futures* has been proposed as a method for
desynchronizing long computations in Lustre \[139\]. A future is a proxy for a result that is initially unknown but becomes known at a later time and can be computed in parallel with other computations.

The ability to observe the passage of physical time in CPSs is integral to their real-time behavior. The modeling of physical time in the synchronous framework is counter-intuitive because the synchrony hypothesis stipulates that all computations must complete instantaneous, i.e., in zero physical time. This issue has been discussed at length by Bourke and Sowmya \[140\] and they suggest the counting of global ticks, use of external timers or intervals, and their proposed delay statement. However, the semantics of delays during preemption and suspension and the preservation of the sampled inputs during a delay is unclear. Asynchronous languages do not have this timing issue and C-based languages with constructs for specifying real-time delays include TimeC \[141\] and Real-Time Concurrency (RTC) \[142\]. Others include the Time-Constrained Event Language (TCEL) \[143\] and the PTIDES \[144\] programming model.

2.1.3 Mixed-Criticality Systems

There is an increasing trend to integrate tasks (or threads in the parlance of synchronous languages) with different requirements into the same system. The requirements may be a combination of, e.g., safety, timing, functionality, and security. The criticality of a task is the level of assurance required against its failure to meet its requirements \[145\]. For example, the IEC 61508 standard \[5\] defines five levels of assurances. A system is called mixed-criticality \[145\] if it has tasks of different criticalities. It is important that the failure of a low criticality task does not cause the failure of a high criticality task. Without such guarantees, all tasks would have to be implemented to the highest criticality, resulting in the over-provisioning of resources. Thus, an important research problem is the efficient scheduling of mixed-criticality tasks such that all tasks meet their requirements. In this thesis, we concentrate on the timing requirements of mixed-criticality systems.

Mixed-criticality systems are typically implemented with real-time tasks on top of a real-time operating system (RTOS) \[146\]. Many scheduling algorithms exist that ensure task deadlines are respected and resources (e.g., processor time, memory, or peripherals) are allocated fairly \[147, 148, 149, 150, 151\] among tasks. Since the introduction and formalization of the mixed-criticality scheduling problem by Vestal \[145\], several studies on uni-processor \[152, 153, 154, 155\] and multi-processor \[156, 157, 158\] systems have emerged. The LInux Testbed for MUlti-Processor Scheduling in Real-Time systems (LITMUS$^RT$) \[159\] is an RTOS designed
2.1 Parallel Programming of CPSs

specially to support mixed-criticality tasks. Partitioned operating systems [160] can partition system resources among groups of tasks to help isolate the hard real-time tasks from the soft real-time tasks. The FlexPRET processor [161] allows the scheduling of mixed-criticality tasks at the hardware level. In Vestal’s formulation [145], the worst-case execution time (WCET) of each task is estimated at the task’s assigned criticality and at all lower criticalities. The assumption is that a task’s estimated WCET is longer, i.e., more conservative, when the task’s criticality is higher. A mixed-criticality system starts executing at its lowest level of criticality by scheduling tasks with their lowest criticality WCETs. If a task exceeds its scheduled time, then the tasks that match the system’s criticality are preempted and dropped. This frees the processor to execute the higher criticality tasks. The system’s criticality is elevated by one level and the remaining tasks are scheduled using their WCETs at the elevated criticality. This process is repeated every time a task exceeds its scheduled time until the system reaches the highest criticality. This strategy, however, causes lower criticality tasks to execute sporadically. A higher criticality task could in theory prevent all the lower criticality tasks from executing. Static [154, 162] and dynamic [145, 152, 153, 156, 158, 163, 164, 165] scheduling strategies have been proposed to ensure that the highest criticality tasks always meet their deadlines. Many of the scheduling strategies are based on earliest deadline first (EDF) because it provides optimal scheduling on preemptive uni-processors. However, frequent task preemptions can add significantly to overheads and affect task schedulability [166].

To alleviate the sporadic execution of low criticality tasks, Su et al. [155, 157] propose an early-release EDF (ER-EDF) scheduling algorithm that guarantees a minimum execution time for all low criticality tasks. The scheduling modifies the periods of the low criticality tasks based on the processor’s runtime utilization. Each low criticality task has a set of recurring early-release times that specifies when it can be released earlier than usual when enough slack has been reclaimed. The results show improved task schedulability and system utilization, but the generation of optimal early-release times was not described. Closely related is the Zero-Slack QoS-based Resource Allocation Model (ZS-QRAM) [163] that aims to maximize system utilization under all operating conditions. ZS-QRAM subsumes earlier work on Zero-Slack (ZS) [152, 164, 167] by supporting the scheduling of criticality and utility-based tasks. The scheduling of criticality-based tasks takes priority over utility-based tasks to ensure that the critical tasks always meet their deadlines. Utility-based tasks use marginal utility to quantify the benefits of allocating extra resources to them. The initial allocation of resources to each task
is performed iteratively, guided by their marginal utility, until all resources are exhausted. At runtime, if a utility-based task exceeds its nominal execution time, then the tasks with lower utility are degraded by extending their period. The time-triggered scheduling work of Goswami et al. [168] consider mixed-criticality systems comprised of safety-critical and time-critical tasks. In their setting, safety-critical tasks have stability and performance constraints, while time-critical tasks only have deadline constraints.

The synchrony hypothesis of synchronous languages requires all tasks to be hard real-time and, therefore, do not support mixed-criticality. In Giotto [123], only the timeliness of task synchronization (called jitter) can be specified. Mixed-criticality synchronous tasks can be emulated by an appropriate scheduling algorithm. The work of Baruah [154] is the first attempt at generating efficient (static) mixed-criticality schedules for multi-rate synchronous programs on uni-processor systems. Baruah applies a previous result [162] to the synchronous setting. Each task is assigned a high or low criticality level and statically scheduled using its low criticality WCET. At runtime, if a high criticality task exceeds its low criticality WCET, then all the low criticality tasks are discarded immediately. As mentioned previously, this causes low critical tasks to execute sporadically. This is undesirable for control-related tasks because sporadic delays are introduced and can cause system instability [115]. Baruah [169] also presents a multi-processor scheduling approach, but only for tasks that execute at the same rate.

2.2 Predictable Hardware Architectures

A program’s timing behavior is influenced directly by the hardware architecture it executes on. Chapter 1.3 described common processor features (e.g., pipelining, superscalar and out-of-order execution, caches, and branch prediction) that improve average-case performance but degrade time predictability. Thus, choosing a suitable hardware architecture is critical. The PRedictability Of Multi-Processor Timing (PROMPT) philosophy [21, 170] advocates the co-design of hardware and software. Each program is analyzed to match it to a suitable hardware architecture. Each component in the architecture should be timing compositional with repeatable behavior. Similarly, the Precision Timed (PRET) philosophy [20, 171] advocates for the use of time predictable architectural features with the ability to control timing through the instruction set architecture (ISA). Although this thesis does not contribute any innovations to architectural design, we review the landscape of predictable architectures for completeness.
2.2 Predictable Hardware Architectures

2.2.1 General-Purpose Processors

We begin by reviewing predictable designs for single-core processors. The Multiple Active Context System (MACS) processor [172] interleaves the multi-cycle execution of independent hardware threads to eliminate pipeline hazards. Each thread accesses its own memory bank to avoid inter-thread interferences. The Microprogrammed Coarse Grained Reconfigurable Processor (MCGREP) [173] has a predictable two-stage pipeline with no cache. The processor’s ISA is defined by a set of customizable microcodes. The compiler optimizes the program’s execution by identifying groups of frequently used machine instructions that can be replaced by a single custom instruction. Platmos [174] is a dual-issue processor that exposes its timing behavior at the ISA level. Processors based on the PRET philosophy include the PRET ARM (PTARM) [31] and the FlexPRET [161] processors. FlexPRET extends PTARM, described in Chapter 1.4.2, to support an arbitrary number of hardware threads with customizable execution frequencies. For multi-cores [18, 19], a greater range of architectural features must be considered [175, 176, 177]. Examples include the use of: partitioned caches [178], software-controlled memories called scratchpads [22] and their allocation schemes [179, 28, 29, 180, 30], transactional memory [181], predictable shared bus arbitration [24, 182, 183, 184, 185], and WCET-aware compilation techniques [186, 187, 188, 189]. The Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (MERASA) multi-core processor [189] supports one hard real-time and three non-real-time threads on each core. Instructions from the hard real-time thread are executed at the highest priority. Hard real-time threads access scratchpads for predictability, while non-real-time threads access caches for performance. Lastly, an analyzable memory controller is used to arbitrate shared bus requests from the cores. For Java programs, there is the Java Optimized Processor (JOP) [25] and its multi-core variant [190].

The ISA serves as an interface between the programmer and the processor. The instructions of the ISA are decoded by the processor into various machine instructions, which are responsible for setting the control signals inside the processor. The PRET philosophy seeks to expose the functionality and timing of the processor through the ISA. Another interesting approach is the No Instruction Set Computer (NISC) architecture [191] that has been used in the hardware/software co-design of real-time systems [192]. The NISC architecture discards the notion of an ISA and exposes the processor’s control signals directly to the compiler. One machine instruction per clock cycle is needed for correct processor operation. Thus, it is imperative for the compiler to create cycle-accurate machine code. Consequently,
the timing behavior of all functional units in a NISC processor must be statically known, i.e., time predictable [193, 194]. In addition, the NISC framework supports the customization of a processor’s datapath by swapping functional units. To achieve this modularity without having to redesign the compiler, all functional units must be timing composable.

### 2.2.2 Reactive Processors

The execution of synchronous programs can be accelerated by reactive processors [195, 196], which have hardware support for signal resolution, concurrency, preemptions, and global tick synchronization. A key feature is their ability to execute programs in a time predictable manner. Single-core multi-threaded reactive processors include the Kiel Esterel Processor (KEP) [196] and Simultaneous multiThreaded Auckland Reactive Processor (STARPro) [197]. Reactive multiprocessors include the Embedded MultiProcessor supporting Esterel Reactive Op-eRations (EMPEROR) [198] and HybriD Reactive Architecture (HiDRA) [199]. However, these reactive processors and associated compilers do not support the execution of host functions written in a host language, such as C. As a compromise between efficiency and host language support, a general purpose processor can be patched with a reactive functional unit to accelerate the execution of synchronous constructs. The Auckland Reactive PRET (ARPRET) [200] processor is a patched Xilinx MicroBlaze processor tailored for executing PRET-C. Java-based reactive processors include the Reactive JOP (RJOP) [201], Tandem Processor JOP (TP-JOP) [202], and GALS Heterogeneous Multiprocessor (GALS-HMP) [203].

### 2.3 Static Timing Analysis

Thus far, we have reviewed the parallel programming of CPSs and the design of time predictable architectures. Recall that the time predictability of a CPS is a function of its software and hardware [8, 9, 204]. For certification purposes, worst-case timing analysis is needed to guarantee that the program’s worst-case timing requirements are met [170]. Thus, it is necessary to compute the program’s worst-case execution time (WCET). As discussed in Chapter 1.7, measurement-based WCET analysis [205, 206, 207, 208, 209] can provide quick, but unsafe, timing estimates. Timing analysis for specific execution paths in the program can be performed using scenario-based timing analysis [210, 211]. However, we are interested in computing safe WCET estimates for every global tick in a synchronous program using static WCET analysis [9, 44]. This computation is known as worst-case re-
2.3 Static Timing Analysis

action time (WCRT) analysis [60, 212]. This section reviews existing techniques for performing static WCET and WCRT analysis.

2.3.1 WCET Analysis

In static timing analysis, the execution time of all the paths through the program’s control-flow graph (CFG) is computed. This requires the program’s CFG to be constructed precisely. Complex compiler optimizations (e.g., code transformation, dead code elimination, and instruction rescheduling) can obfuscate the program’s control-flow, making analysis difficult. Thus, it is common to only allow basic compiler optimizations so as to retain the original control-flow. To explore all the CFG paths of a single-threaded program, implicit and explicit path-enumeration techniques can be used. Implicit path-enumeration can be performed with integer linear programming (ILP) [213], where the CFG is modeled as a system of integer constraints. Using a precise timing model of the hardware architecture, a constraint is created for every edge in the CFG to model the preceding node’s execution time. By solving the constraints, the program’s CFG is implicitly explored and the maximum solution is the program’s computed WCET. The complexity of ILP is known to be NP-hard [214]. If the CFG contains conditional branches that test a common set of variables, then some of these branches are likely to execute together. Thus, additional constraints are needed to track the values of the variables to identify the infeasible CFG paths [215]. If the CFG contains loops, then the constraints along the loop need to be multiplied by the maximum number of loop iterations. The number of iterations can be determined by loop analysis [216].

Explicit path-enumeration can be performed with model checking [217] or reachability. Both techniques explore all the possible execution paths in a program’s CFG, i.e., the complete reachable state-space. Before the CFG is explored, every edge in the CFG is annotated with the preceding node’s execution time. The CFG is explored and the execution times along each path are summed up to compute the path’s total execution time. During the CFG exploration, the iterations of each loop and the values of each variable can be tracked and analyzed. Cassez and Bechennec [218] propose the use of program slicing to automatically find the variables and associated CFG nodes that need to be tracked during timing analysis. For model checking, the computed WCET is retrieved by verifying that the execution time of all paths is less than the user’s guess for the WCET. The user’s guess can be refined by using a binary search. A binary search is not required for reachability because every path through the program is explored and the longest computed WCET is returned. Hence, reachability scales better than model check-
ing, but both techniques experience the well known *state-space explosion* problem when large and complex CFGs are analyzed.

For sufficiently complex CFGs and timing models, the time needed to compute the WCET using ILP, model checking, and reachability can be substantial. Abstract interpretation [59] can be used to simplify the program’s CFG and the timing models of various hardware components (e.g., the pipeline [219] and memory [220, 221]) to create a less precise (but still correct) model that is faster to analyze [222]. Abstractions need to be chosen carefully to minimize any degradation on the precision of the computed WCET. Programmers can help simplify the program’s CFG by deliberately writing code without branching statements, i.e., single-path programming [223].

ILP, model checking, and reachability can be used to statically analyze parallel programs deployed on parallel hardware architectures. Unfortunately, the methods developed for single-core analysis are not readily transferable to the multi-core setting. This is because multi-core execution presents the following challenges:

**Inter-core interference:** Cores that simultaneously access the same hardware resources (e.g., memory and bus) will interfere with each other if the resource can only service one request at time. The analysis must consider the delays that a core can experience whenever it accesses a shared resource.

**Thread-level parallelism:** Threads that execute simultaneously on separate cores will have overlapping execution times. The analysis must not add the threads’ execution times together, but instead take the maximum of the threads’ execution times as the overall execution time. Programs with nested threads are more challenging to analyze.

**Software-based synchronization:** Threads can achieve mutual exclusion by using software-based synchronization constructs (e.g., locks). These constructs cause the threads (and their cores) to block for a variable amount of time. The analysis must identify synchronization points in the CFG to analyze the blocking times accurately for precise analysis.

**Thread scheduling:** Threads can be scheduled globally or locally across the cores. The analysis must consider the scheduling behavior on each core (e.g., thread ordering, context-switching, overheads, and additional interferences).

Because these challenges stem from inter-core interactions, the cores must be analyzed jointly for precise WCET computation. For ILP, Yan and Zhang [224] assume all cores have private level-one instruction caches that connect directly to a shared
level-two instruction cache. Independent threads were assumed to execute on each core. The level-one caches were analyzed and any misses would require access to the shared level-two cache. The WCET is computed by taking into account the cost of encountering misses in the cache hierarchy. Kelter et al. [225] assume a more complex architecture where cores are connected to the shared level-two cache by a shared TDMA bus. Kelter et al. exploit the time predictability of TDMA to create an abstraction that minimizes the unrolling of loops [216] when calculating their execution times. Chattopadhyay et al. [226] observe that only threads with overlapping execution times can evict each other’s instructions from the shared level-two cache. This reduces the number of level-two cache misses and a better WCET estimation can be computed. Chattopadhyay et al. [227] assume a more complex architecture where branch prediction is enabled and timing anomalies can occur in the pipeline. Results show that the proposed approach can compute good WCETs in reasonable time for their selected benchmarks. For model checking, Lv et al. [228] assume a simpler architecture where the pipeline is free of speculation and timing anomalies. Abstract interpretation is used to analyze each core’s cache behavior and timed automata are created to model the shared TDMA bus and when bus accesses occur because of cache misses. The models are explored to compute each program’s WCET. Gustavsson et al. [229] create more detailed timed automata of the pipeline and instruction and data caches, but assume that all cores connect directly to shared memory without a shared bus. Lock-based mutual exclusion can be modeled in the timed automaton of the program’s CFG. Preliminary results with a small program show that the approach becomes intractable when only four cores are analyzed.

It is clear that research on WCET computation has focused on analyzing increasingly complex hardware features, but for simple multi-threaded programs only. For example, most approaches have not considered 1) the execution of nested threads, 2) the synchronization of threads due to software-based mutual exclusion, and 3) the effects of thread scheduling. Liang et al. [230] provide a method for computing the WCET of a program with threads that communicate using message passing. The analysis is performed on a message sequence chart, which captures a particular execution scenario of the program. A strategy for creating all the necessary message sequence charts to ensure a safe WCET computation was not provided. Ozaktas et al. [231] analyze parallel programs with software-based synchronization constructs (barriers and locks). All threads are assumed to execute the same code and ILP is used to compute the WCET. The work does not discuss the analysis of nested threads or scheduling overheads. Gustavsson et al. [232]
propose a simple parallel programming language that provides locks for shared memory communication. The execution time of each language construct is defined by its abstract execution semantics. The WCET is computed by an abstract execution of the program. Analyzing the WCET of a program deployed on an RTOS is challenging because analyzing an RTOS is itself a challenging task [233, 234, 235]. Many RTOSs have been studied for timing analyzability with mixed successes: the RTEMS, OSE kernel, L4 Kernel, and TUWien by Lv et al. [236], OSE kernel [237], μC/OS-II [238], and MERASA RTOS [32]. Common issues with analyzing RTOSs include: difficulty in analyzing programming features such as function pointers and indirect jumps, lack of knowledge on RTOS execution states, non-deterministic preemptions due to interrupts, and interferences between the program’s tasks and the RTOS’s scheduler.

Using the computed WCET to improve the design of a CPS is an objective that has not been carefully addressed. A key issue is the difficulty in relating the low-level timing analysis results back to the high-level source code written by the programmer [23, 239]. Another issue is that WCET analysis can take a long time to complete, making the debugging of timing violations a tedious exercise. However, Harmon et al. [239] have developed a code editor that displays immediate feedback of rough WCET estimates for blocks of code. WCET-aware compilers [23, 240] are being developed to generate code that is optimized for WCET performance. WCET analysis has also be used to help designers reduce a system’s energy consumption [241, 242].

2.3.2 WCRT Analysis

As described in Chapter 2.1.2, synchronous compilers typically generate sequential programs with internal state variables that decide the program’s next execution state. If the analysis does not know how the internal state variables are used, then many infeasible paths may be explored [243]. Other compilers generate concurrent threads that correspond to those in the original synchronous program. In each global tick, the threads are scheduled to execute their respective local tick. Thus, a global tick consists of executing a set of local ticks. Due to conditional branches and loops, there may be sets of local ticks that can never execute in the same global tick [244, 245]. Finding the feasible sets is known as the tick-alignment problem [246]. Note that infeasible paths may exist in each thread, e.g., from programmer written code. Thus, for precise WCRT computation, the analysis must prune away infeasible tick-alignments and infeasible paths.

The WCRT of a synchronous program can be described in a modular and hier-
archival manner using WCRT algebra [212]. The WCRT computation is described using max-plus algebra on a timing matrix. Various algorithms can be used to solve the algebraic equations and employ various abstractions to trade-off precision with analysis time. ILP, model checking, and reachability have been used to compute the WCRT of (multi-threaded) synchronous programs deployed on single-cores. For ILP, Ju et al. [245] analyze synchronous programs that have been compiled into sequential programs. The internal state variables are taken into account in a special tick transition automata to prune infeasible paths and infeasible tick-alignments. This work was extended [247] to display the worst-case execution path of Esterel programs back to the user for performance debugging. ILPc [248] analyzes synchronous programs that have been compiled into concurrent threads. The analysis starts by assuming all tick-alignments are possible. If the ILP solution corresponds to an infeasible tick-alignment, then the WCRT is recomputed with extra constraints that ignore this particular tick-alignment. This iterative refinement continues until a WCRT is computed for a feasible tick-alignment. Thus, by avoiding the need to generate all the constraints needed to prune all the infeasible tick-alignments, ILPc has much shorter average runtimes than previous ILP methods. Raymond et al. [249] propose a similar iterative approach that employs model checking to refute the worst-case path found by the ILP solution.

Model checking and reachability have the advantage that they can show the sequence of global ticks that lead to the WCRT. This traceability is useful when debugging timing violations. For model checking, TAXYS [250] is a framework for verifying the timing properties of an Esterel program against a non-deterministic model of the environment. Real-valued clocks are used, which leads to exponential complexity. TAXYS, however, does not compute the program’s WCRT. Roop et al. [244] analyze synchronous programs that have been compiled into concurrent threads. An automaton is created for each thread. To prune infeasible tick-alignments, an automaton is created to model the barrier synchronization at the end of each global tick. This work was extended [251] with variable tracking to prune infeasible paths and more infeasible tick-alignments for more precise WCRT computation. For reachability, Boldt et al. [60] compute the WCRT of each thread separately and sum them together for the program’s WCRT. The WCRT of each thread is computed by reachability on the thread’s CFG. This high level of abstraction reduces the analysis complexity to be linear with respect to the number of threads. However, the abstraction assumes that the worst-case paths of all threads will occur in the same global tick, which may lead to large over-estimations. Kuo et al. [252] compute the WCRT more precisely at a finer granularity by taking
tick alignment into account. Variables are tracked to prune infeasible paths. The state-space is reduced by simplifying the CFG in two ways: sequential CFG nodes are merged into a single node, and branches are merged into a single path.

For multi-cores, Ju et al. [47] propose a reachability-based algorithm to compute the WCRT. The synchronous program is statically partitioned over the cores and a sequential program is generated for each core. Blocking communication is used to resolve signal statuses between the cores. It is likely that only a subset of Esterel programs can be compiled correctly and, therefore, analyzed. The analysis takes tick-alignment into account by analyzing the internal state variables generated by the compiler. Every core has private level-one instruction cache and connect to main memory via a shared TDMA bus. For safe analysis, only the worst-case bus delay is used. Unfortunately, results for the precision of the computed WCRT were not discussed. Moreover, the execution performance of the multi-core compilation approach was not discussed.

2.4 Discussion

This chapter has presented a snapshot of the current efforts in developing time predictable CPSs that integrate mixed-criticality tasks on parallel hardware architectures. Many of the existing attempts at providing deterministic parallelism have used concepts that already feature in synchronous languages. The C-based synchronous languages have much to offer to embedded programmers in terms of deterministic concurrency and formally verifiable implementations, but lack support for parallel execution. When execution time bounds are needed to satisfy the synchrony hypothesis in a time-triggered execution model, the synchrony hypothesis is incompatible with the notion of mixed-criticality, where tasks can be scheduled at different levels of assurances. Static timing analysis techniques have made great strides in analyzing low-level hardware features and developing suitable abstractions. It is now the time to incorporate additional program contexts to enable the analysis of software-based synchronization patterns and runtime behaviors. This will greatly help the complex analysis of RTOSs. Making use of the computed WCETs and WCRTs to facilitate design space exploration (e.g., energy consumption, code size, execution time, thread allocation, and hardware architecture) is still an open research topic.

This thesis tackles the lack of a C-based synchronous parallel programming language that offers good time predictability and parallel execution performance. First, the ForeC language is presented in Chapter 3 for the deterministic parallel
2.4 Discussion

programming of multi-cores. Second, the ForeCast static timing analyzer is presented in Chapter 5 that offers a comprehensive analysis of software-based synchronizations and scheduling behaviors. Third, the ForeMC framework is presented in Chapter 7 that extends the synchronous framework towards mixed-criticality.
A cyber-physical system (CPS) is an inherently parallel and reactive system with strict timing and functional requirements. Execution platforms have evolved from single-cores to multi-cores. Hence, all the synchronous languages designed earlier (e.g., Esterel [45], Lustre [116], Signal [117], Esterel C Language [51], Reactive Shared Variables [95], and PRET-C [50]) must be reinvented to address the challenges raised by multi-cores. We believe synchronous languages are great for writing concurrent programs and propose the ForeC language dedicated to multi-cores. ForeC is synchronous to benefit from the formal deterministic semantics and extends C to benefit from the control and data structures available in C. A key goal of ForeC is in providing deterministic shared variable semantics that is agnostic to scheduling. This goal is essential for the reasoning and debugging of parallel programs. This chapter illustrates ForeC with the UAV running example described in Chapter 1.2. The formal semantics of ForeC is detailed and important proofs concerning program reactivity and determinism [253, 254] are provided.

3.1 Overview

ForeC is a synchronous language that extends a subset of C (see Chapters 1.6 and 3.1.7) with a minimal set of synchronous constructs. Figure 3.1 gives the extended syntax and Table 3.1 summarizes the semantics. A statement (st) in ForeC can be a traditional C statement (c_st, e.g., a function call, if-else, while-loop, or variable assignment), or a barrier (pause), fork/join (par), or preemption (abort) statement. Using the sequence operator (;), a statement in ForeC can be an arbitrary composition of other statements. Like C, extra properties can be specified for variables using type qualifiers. A type qualifier (tq) in ForeC is a traditional C type qualifier (c_tq), an environment interface variable (input and output), or a shared variable amongst threads (shared). The input, output, and
The ForeC Language

Statements: \[ st ::= c_{st} \mid \text{pause} \mid \text{par}(st, st) \]
\[ | \text{weak? abort st when immediate? (exp)} \]
\[ | st; st \]

Type Qualifiers: \[ tq ::= c_{tq} \mid \text{input} \mid \text{output} \mid \text{shared} \]

Figure 3.1: Syntactic extensions to C.

| input: Type qualifier to declare an input variable, the value of which is updated by the environment at the start of every global tick. |
| output: Type qualifier to declare an output variable, the value of which is emitted to the environment at the end of every global tick. |
| shared: Type qualifier to declare a shared variable, which can be accessed by multiple threads. |
| pause: Pauses the executing thread until the next global tick. |
| par(st, st): Forks two statements st as parallel threads. The par terminates when both threads terminate (join back). |
| weak? abort st when immediate? (exp): Preempts its body st when the expression exp evaluates to a non-zero value. The optional weak and immediate keywords modify its temporal behavior. |

Table 3.1: ForeC constructs and their semantics.

shared type qualifiers precede the C type qualifiers in variable declarations.

We illustrate the ForeC language with the UAV example shown in Figure 3.2 as a block diagram of tasks. The UAV consists of two parallel tasks called Flight and Avoidance. The Flight task is comprised of two parallel tasks called Navigation and Stability. The Navigation task localizes the UAV with on-board sensors, updates the flight path, and sends the desired position to the Stability task. The Stability task controls the flight surfaces to ensure stable flight to the desired position. The Avoidance task is comprised of two parallel tasks called FindL and FindR. These tasks use on-board sensors to detect obstacles around the UAV and sends collision avoidance data to the Navigation task.

Figure 3.3 is a ForeC implementation of the UAV example given in Figure 3.2. Figure 3.4 is a possible execution trace of Figure 3.3 to help illustrate the execution of ForeC programs. Chapter 1.5.1 described the execution behavior of synchronous programs. To recap, the threads of a synchronous program execute in lock-step to the ticking of a global clock. In each global tick, the threads sample the environment, perform their computations, and emit their results to the environment. When a thread completes its computation, we say that it has completed its local tick. When all the threads complete their local ticks, we say that the program has completed its global tick. In Figure 3.4, the first three global ticks are demarcated
3.1 Overview

Figure 3.2: Tasks of the UAV.

along the lefthand side.

In Figure 3.3, the UAV program starts with the inclusion of a C header file (line 1) for the functions used in the program and the global variable declarations (lines 2–3) to interface with the environment. Line 2 declares input variables to capture sensor readings. Input variables are read-only and their values are updated by the environment at the start of every global tick. Line 3 declares output variables to hold actuation commands for the flight motors and surfaces. Output variables emit their values to the environment at the end of every global tick. Input and output variables can only be declared in the program’s global scope. The lefthand side of Figure 3.4 shows the sampling of inputs and emission of outputs at the start and end of each global tick, respectively.

Like traditional C programs, the function main (line 5) is the program’s main entry point and serves as the initial thread of execution. Lines 6–7 declare variables that can be shared amongst threads (see Chapter 3.1.3). In Figure 3.4, the states of the shared variables are given inside solid round boxes at specific points in the execution trace. Line 6 declares a shared variable obst to store the distance and angle of the closest obstacle as an encoded integer. Line 7 declares a shared variable newPos to store the UAV’s desired position.

On line 8, the par statement forks the Flight (line 10) and Avoidance (line 19) functions into two parallel child threads. We refer to the threads by their function names, e.g., the Flight and Avoidance threads. The forking of threads is repre-
```c
#include <uav.h>
int pos1, pos2, proxl, proxr; // Inputs.
int motors=0, flaps=0; // Outputs.

void main( void ) {
  void obst=0 combine new with min;
  void obst newPos=0;
  par( Flight( newPos,&obst ), Avoidance( &obst ) );
}
void Flight( int *newPos, int *obst ) {
  par( Navigation( newPos, obst ), Stability( newPos ) );
}
void Navigation( int *newPos, int *obst ) {
  while ( 1 ) { *newPos=plan( pos1, obst );
    pause; }
}
void Stability( int *newPos ) {
  while ( 1 ) { motors=thrust( pos2, newPos ); flaps=angle( pos2, newPos );
    pause; }
}
void Avoidance( int *obst ) {
  while ( 1 ) { par( { *obst=find( proxl ); }, // Thread FindL.
                     { *obst=find( proxr ); } ); // Thread FindR.
    pause; }
}
int min( int th1, int th2, int pre ) { if ( th1<th2 ) return th1; else return th2; }
```

Figure 3.3: Example ForeC program for the UAV running example.

Figure 3.4: Possible execution trace for Figure 3.3.
3.1 Overview

presented in Figure 3.2 as triangles. On line 11, the Flight thread forks two more parallel child threads, Navigation (line 13) and Stability (line 16), creating a hierarchy of threads. The par statement can also fork blocks of code, e.g., line 20 forks the FindL and FindR threads. The par is a blocking statement and terminates only when both its child threads have terminated and joined together. The joining of threads is represented in Figure 3.4 as inverted triangles.

After the Navigation, Stability, FindL, and FindR threads have forked, they start executing their respective body. For example, the Navigation thread enters the while-loop (line 14) and computes a new desired position. Next, the pause statement pauses the thread’s execution (line 15), acting as a synchronization barrier. In Figure 3.4, the pause statements are shown as black rectangles and the program completes a global tick when all the threads pause. This is indicated by dotted horizontal lines across the pause statements.

Every time a thread starts its local tick, it creates local copies of all the shared variables that its body accesses (reads or writes). The shared variables declared in the program remain distinct from the threads’ local copies. When a thread needs to access a shared variable, it accesses its local copies instead. Thus, the changes made by a thread cannot be observed by others, yielding mutual exclusion and thread isolation. Moreover, only sequential reasoning is needed within a thread’s local tick. In Figure 3.4, the states of a thread’s copies are shown inside dotted round boxes throughout the execution trace. For example, when the Navigation thread starts its first local tick, it has a copy of obst and newPos (values equal to 0). When its local tick ends, its copy of newPos has been set to 56.

To enable thread communication, the copies of each shared variable are automatically combined into a single value when the threads join and when the global tick ends. This is achieved by a programmer-specified combine function. In Figure 3.3, the combine function for obst (line 6) is min (line 23), specified by the combine clause, which returns the closest obstacle. The combine clause also specifies that only the copies with new values are combined. In global tick one of Figure 3.4, the FindL and FindR threads set new values (2 and 3) to their copies of obst. When these threads join, the new values are combined to 2 and assigned to their parent thread Avoidance. Meanwhile, the Navigation thread only reads its copy of obst. Thus, when global tick one ends, the value of the shared variable obst is set to 2. Had there been more copies with new values, then these copies would have been combined and assigned to obst before the next global tick started. We say that the shared variables are resynchronized at the end of each global tick. In Figure 3.4, the resynchronized values, called pre values, are shown
inside solid round boxes. The shared variables start each global tick with their pre values. Because threads can only modify their copies, the shared variables maintain their pre values throughout the global tick. For the first global tick only, the pre of a shared variable is its initialization value.

Appendix B describes more examples of combine functions and how more than two copies are combined. The following sections elaborate on the details of local and global ticks, fork/join parallelism, shared variables, and preemption.

### 3.1.1 Local and Global Ticks

We say that a thread completes its local tick when it pauses, terminates, or forks at least one thread that completes their local tick without terminating. For example, in Figure 3.3, the Avoidance thread starts its first local tick by forking the child threads FindL and FindR (line 20). Assuming that the find function does not pause, both child threads complete their local tick by terminating. After the child threads join, the Avoidance thread reaches a pause (line 22) and completes its first local tick. A program completes its global tick when all its threads have completed their respective local ticks. At the next global tick, the paused threads start their next local tick from their respective pauses. For brevity, we shorten “global tick” into “tick” and use “local tick” as before.

### 3.1.2 Fork/Join Parallelism

The par statement enables the forking of parallel threads. We articulate some well known terminology related to parallel programs to be used consistently later. The parent thread is the thread that executes the par statement to fork its child threads. The parent thread is also the ancestor of its child threads and their nested child threads. Child threads forked by the same par statement are siblings. Because the par is a blocking statement, threads always execute sequentially with respect to their ancestors. Threads that are not ancestors of each other are relatives and can execute in parallel. The thread genealogy of a program can be determined statically by inspecting the program’s control-flow. Figure 3.5a shows the thread genealogy of the UAV program. Each node is a thread and arrows are drawn from the children to their parent thread. Figure 3.5b exemplifies the thread genealogy.

### 3.1.3 Shared Variables

All variables in ForeC follow the scoping rules of C. By default, all variables are private and can only be accessed (read or write) by one thread throughout its scope.
To allow a variable to be accessed by multiple threads, it must be declared as a *shared* variable by using the `shared` type qualifier. Thus, shared variables are easy to identify and any misuse of private variables are easy to detect at compile time.

ForeC defines the semantics of shared variables for thread-safe sharing without the need to use mutual exclusion constructs. The goal is to provide a deterministic shared variable semantics that is agnostic to scheduling, which is essential for the reasoning and debugging of parallel programs. Within each tick, the accesses to a shared variable from two threads may occur in sequence or in parallel:

**Definition 1.** *Accesses from two threads are in sequence if both threads are not relatives or if the accesses occur in different ticks.*

**Definition 2.** *Accesses from two threads are in parallel if both threads are relatives and the accesses occur in the same tick.*

Improperly managed parallel accesses to a shared variable can cause race conditions, leading to non-deterministic behavior. For example, two parallel writes to a shared variable can non-deterministically and partially overwrite each other’s value. A parallel read and write to a shared variable can result in the read returning the variable’s value before, during, or after the write has completed. Table 3.2 reviews the solutions that exist for enforcing mutual exclusion on shared variables, usually by interleaving parallel accesses into a sequence. A set of parallel accesses can be interleaved in many ways (influenced by the programmer, compiler, and runtime system) and relying on a particular interleaving for correct program behavior is brittle and error prone.

We propose a shared memory model that permits shared variables to be accessed deterministically in parallel, without needing the programmer to explicitly use mutual exclusion. The goals of the model are:

**Isolation:** Provide isolation between threads to enable the local reasoning of each
**Programming Constructs:** These are constructs written in the host language to provide mechanisms for the programmer to achieve mutual exclusion. Examples include: locks, monitors, transactional memory, message passing, and parallel data structures. Using these constructs correctly can be tedious and error prone for large programs and may lead to other errors [38, 39, 79], e.g., deadlocks, starvation, or priority inversion.

**Language Semantics:** The language semantics can have a memory model that defines how threads interact through memory, what value a read can return, and when the value of a write becomes visible to other threads. Although the memory model can prevent race conditions, it may only be suitable for a few types of applications. Examples include: synchronous languages [41], PRET-C [50], Synchronous C [52], SharC [90], Deterministic Parallel Java [91], SHIM [92], ΣC [93], concurrent revisions [94], and Reactive Shared Variables [95].

**Static Analysis:** A compiler or static analyzer can identify and alert the programmer regarding the race conditions in their program (e.g., Parallel Lint [96]) and may try to resolve them by serializing the parallel accesses for the programmer (e.g., Sequentially Constructive Concurrency [97]). However, programmer guidance is needed for race conditions that cannot be resolved.

**Runtime Support:** Programs are can be executed on a runtime layer that dynamically enforces deterministic execution and memory accesses. Examples include: dOS [98], Grace [99], Kendo [100], CoreDet [101], Dthreads [102], DOMP [103], and DetMP [104]. However, understanding the program’s behavior at compile time remains difficult because the determinism is only enforced at runtime.

**Hardware Support:** Parallel accesses can be automatically detected and resolved by the hardware, preventing race conditions from happening. Examples include: Ultracomputer’s combine hardware [105] and certain shared bus arbitration (e.g., round-robin, TDMA, and priority). However, the timing of the parallel accesses affects how they are interleaved.

| Table 3.2: Existing solutions for avoiding race conditions. (Reproduction of Table 2.1 in Chapter 2.1.1 for the reader’s convenience.) |
3.1 Overview

thread. That is, the execution of a thread’s local tick can be understood by
only knowing the values of variables at the start of the thread’s local tick.

Determinism: Ensure deterministic execution regardless of scheduling decisions.
This guarantees that deterministic outputs are always generated at the end
of each tick.

Parallelism: Minimize the need to serialize parallel accesses to shared variables.
This maximizes the amount of parallel execution that can occur at runtime,
which is important for improving the program’s performance.

We propose the following mechanisms for achieving parallel constructiveness: All
threads access their own local copies of the shared variables, and these copies are
resynchronized every time threads join and when the tick ends.

3.1.4 Copying of Shared Variables

Every time a thread starts its local tick, it creates local copies of all the shared
variables that its body accesses (reads or writes). When a thread is forked, its
initial copy of a shared variable is created from its parent’s copy if available,
otherwise, from the shared variable’s resynchronized value (pre). A parent thread
that is blocked on a par statement does not create any copies of shared variables
until the par statement terminates. For example, in tick two of Figure 3.4, the main
thread makes no copies. Shared variables can be passed as arguments into threads.
Following the C convention, when a shared variable is passed by value, only its value
is used to initialize the thread’s parameter. A shared variable declared inside a
thread can be shared among its child threads by passing a reference (using a
pointer) into the child threads (e.g., obst on line 8 of Figure 3.3). When a shared
variable is passed by reference into an ordinary function (e.g., obst on line 14),
the function uses the calling thread’s copy of the shared variable.

3.1.5 Resynchronization of Shared Variables

The copies are resynchronized every time the program completes its tick (before
outputs are emitted). Resynchronizing at specific program points ensures that the
semantics of shared variables is agnostic to scheduling. Combine functions must
be stateless, deterministic, associative, and commutative. That is, the combine
function produces the same outputs from the same inputs, regardless of previous
invocations and how the copies are ordered or grouped. The signature of any
combine function is $C : Val \times Val \times Val \rightarrow Val$. The first two input parameters
typedef enum {OK, ERROR, WARN, TERM} State;

input State comm; // Additional input.

... void main(void) {
    shared int obst = 0 combine new with min;
    shared int newPos = 0;
    abort {
        par (Flight(&newPos, &obst), Avoidance(&obst));
    } when (comm == TERM);
    safeDescent();
}

Figure 3.6: Figure 3.3 extended with preemption.

are the two copies to combine. The third input parameter is the shared variable’s pre value. When a par statement terminates, the copies from the terminating child threads are combined and assigned to their parent thread’s copies of shared variables. For example, in Figure 3.4, the Avoidance thread gets a copy of obst every time FindL and FindR terminate. Appendix B describes more examples of combine functions and how more than two copies are combined.

It can be useful to ignore some of the copies when resynchronizing a shared variable. This is achieved by specifying a combine policy that determines what copies will be ignored. The combine policies are new, mod, and all. The combine policy of a shared variable is specified during variable declaration in the combine clause, e.g., combine new with. The new policy ignores copies that have the same value as pre. The mod policy ignores copies that were not assigned a value during the tick. The default policy is all where no copies are ignored. Note that the combine function is not invoked when only one copy remains. Instead, that copy becomes the resynchronized value. Appendix B provides extensive illustrations comparing the behavior of the combine policies.

Often, a shared variable is intentionally used where only one thread writes to it and the other threads read from it. Its combine function would never be invoked if the new or mod combine policy is used. As a shorthand, the combine clause can be dropped from a shared variable’s declaration if 1) the new or mod combine policy is used, and 2) it can be statically proved that parallel threads will not write to it. A proof may not always be possible because some writes to the shared variable may be executed conditionally by if-statements or loops. The compiler can make the conservative assumption that all conditional writes to a shared will always occur. Line 7 in Figure 3.3 is an example of this shorthand.
3.1 Overview

3.1.6 Hierarchical Preemption

Inspired by Esterel [45], the `abort st when (exp)` statement provides preemption [255], which is the termination of the `abort` body `st` when the condition `exp` evaluates to `true`. The condition `exp` must be a side-effect free expression, consisting only of variables and the comparison and logical operators. Preemption can be used to model state machines succinctly. In Figure 3.6, the `main` function of the UAV has been extended to respond to external commands through the input variable `comms` (line 2). The value of `comms` can be `OK`, `ERROR`, `WARN`, or `TERM` (line 1). The `abort` statement on line 7 preempts the execution of all the UAV tasks when `TERM` (termination) is received. The UAV descends safely and the program terminates (line 10). A possible execution trace of the extended program is given in Figure 3.7 with italicized line numbers referring to the line numbers in Figure 3.3.

We now explain the semantics of the `abort` statement. The preemption of the `abort` must be `triggered` before it can be `taken`. Preemption cannot be triggered when the `abort` body executes for the first time (e.g., tick one in Figure 3.7). At the start of each subsequent tick, the condition `exp` is evaluated before the `abort` body can execute. This allows shared variables in the condition to be evaluated with their `stable pre` value. If `exp` evaluates to `true` (a non-zero value following the C convention), then the preemption is triggered and the `abort` statement is terminated. At the start of tick two in Figure 3.7, preemption is triggered because the preemption condition evaluates to `true`. The `abort` statement also terminates if its body terminates normally.

Preemptions in ForeC differ to those in Esterel because Esterel uses signals for thread communication rather than variables. As introduced in Chapter 1.5.1, signals in Esterel are either present or absent in each tick and this information is propagated instantaneously across the threads without delay. Thus, preemptions in Esterel are triggered instantaneously, whereas preemptions in ForeC are triggered with a delay of one tick. Like Esterel [255], the optional `weak` and `immediate` keywords change the temporal behavior of preemptions. Figure 3.8a is an example of an `abort` with the optional keywords commented out. Its execution is summarized in Figure 3.8b. In tick one, the `main` thread sets its copy of `s` to 1 and prints “1”. Next, the threads `t0` and `t1` set their copies of `s` to 2 and 5, respectively. When the tick ends, using the combine policy `all`, the resynchronized value of `s` is 7. In tick two, the `abort`’s preemption is triggered and taken, resulting in “7” being printed. The `weak` keyword delays the `abort`’s ability to take a triggered preemption until its body cannot execute any further. Uncommenting the `weak` keyword in Figure 3.8a results in Figure 3.8c. In tick two, the triggered preemption
The ForeC Language

Figure 3.7: Possible execution trace for Figure 3.6.

```c
shared int s=0 combine all with plus;
int plus(int th1, int th2, int pre) { return (th1+th2); }
void main(void) {
    s=1; printf("%d",s);
    /*weak*/ abort {
        par({s=2;pause;s=3;pause;s=4;},
            {s=5;pause;s=6;pause;s=7;});
    } when /*immediate*/ (s>0);
    printf("%d",s);
}
```

(a) Example code.

(b) Non-immediate and strong `abort`.

<table>
<thead>
<tr>
<th>Tick 1: “1” printed.</th>
<th>Tick 1: “1” printed.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = plus(2,5,0) = 7.</code></td>
<td><code>s = plus(2,5,0) = 7.</code></td>
</tr>
</tbody>
</table>

(c) Non-immediate and weak `abort`.

|------------------------------------------------------|-------------------------------|

(d) Immediate and strong `abort`.

|--------------------------------------------------------------------------------|-----------------------------|

(e) Immediate and weak `abort`.
is taken when both threads complete their local tick, i.e., cannot execute further. Thus, “9” is printed.

The immediate keyword allows preemption to be triggered before the abort body executes for the first time. That is, the preemption condition exp is evaluated immediately when execution reaches the abort. Uncommenting the immediate keyword in Figure 3.8a results in Figure 3.8d. In tick one, the main thread sets its copy of s to 1 and prints “1”. Next, the abort’s preemption condition is evaluated immediately. Intuitively, because “1” was printed for the value of s, the condition s>0 should evaluate to true. The counter-intuitive result of false would occur if the pre of s was used. Thus, when execution reaches an immediate abort, the condition exp is evaluated immediately with the thread’s copies of shared variables. In subsequent ticks, the pre of the shared variables is used. Uncommenting both the weak and immediate keywords in Figure 3.8a results in Figure 3.8e. In tick one, the triggered preemption is taken only when both threads complete their local ticks. Hence, “7” is printed.

The abort statements can be nested to create a hierarchy of preemptions with the outer abort executing before the inner aborts. Thus, the preemption behavior of the outer abort takes precedence over the inner aborts. Figure 3.9 is an example of an immediate and weak abort (line 3) with a nested immediate and strong abort (line 5). In tick one, preemption is triggered for the outer weak abort. The variable x is set to 2 and the inner strong abort preempts immediately without executing its body. Next, x is set to 5 and the outer weak abort takes its preemption when it reaches the pause on line 6. Finally, “5” is printed.

### 3.1.7 Bounded Loops

In addition to the strict C-coding guidelines described in Chapter 1.6, ForeC forbids the use of unbounded recursion of function calls and thread forking to ensure runtime reliability. The synchrony hypothesis requires each tick to execute in
finite time, which means all statements need to have bounded execution times. Unfortunately, loop constructs (for and while) can have unbounded iterations, leading to unbounded execution times. Thus, if a loop construct is used, then the programmer must guarantee that it always terminates or executes a pause after a bounded number of iterations. Guaranteeing that a loop always executes a pause may not be possible when pause statements are enclosed by if-statements. The compiler can make conservative assumptions to prove whether a loop always executes a pause in each iteration. For example, a loop is assumed to always execute a pause in each iteration if its body has at least one statement that always executes a pause. An if-statement is assumed to always execute a pause if both its if–else branches always executes a pause. An abort statement is assumed to never execute a pause. A par statement is assumed to always execute a pause if at least one of its child threads always executes a pause. The compiler can perform structural induction on the program’s control-flow to conservatively prove whether every loop in the program will always execute a pause in each iteration.

Inspired by PRET-C [50], we have extended the syntax of loops to help the programmer write bounded loops, shown in the first column of Table 3.3. The “#n” after the loop header specifies that only up to n iterations can be executed. The second column of Table 3.3 gives the structural translation of each bounded loop.

For the translation of a bounded for-loop, the variable cnt tracks the number of iterations that have executed. The condition (cnt<n) guarantees that only up to n iterations are executed. The bounded while-loop is translated into a bounded for-loop. For the translation of a bounded do-while-loop, the variable first is used to delay the evaluation of cond to the second iteration. This delay emulates the execution behavior of a do–while-loop.

### 3.2 Design patterns

Design patterns [256, 257, 258] are reusable templates that programmers can use to solve recurring problems. Design patterns for parallel programming should offer structured solutions that are easy to understand, produce reliable programs,
usable for any problem size, and give high execution performance. Multiple design patterns can be composed together to solve complex problems. In this section, we present six design patterns [256, 257, 258, 259] commonly used in parallel programming and elaborate on their ease of use in ForeC.

### 3.2.1 Point-to-Point

The point-to-point pattern solves the problem of two threads needing to communicate. An instance of this pattern is given in Figure 3.10a. For this simple pattern, the first thread is the sole writer of a shared variable while the other thread is the sole reader of the shared variable. Hence, communication is one-way. A second shared variable can be used to achieve two-way communication, where the first thread is the sole reader while the second thread is the sole writer. In Figure 3.10a, thread $t_1$ communicates to $t_2$ with the shared variable $s$. Thread $t_2$ communicates to $t_1$ with the shared variable $t$. Note that the communication is delayed until the end of the tick. Notice that mutual exclusion constructs, such as locks, are not needed for thread communication. Hence, concurrency errors [79, 39], such as deadlock, race condition, and atomicity, are not possible in ForeC programs. Also, causality cycles [254] suffered by Esterel do not exist.

### 3.2.2 Broadcast

The broadcast pattern solves the problem of a thread needing to communicate to a group of threads. An instance of this pattern is given in Figure 3.10b. The one-way communication of the point-to-point pattern is extended to allow multiple threads to read from (but not write to) the shared variable. Again, notice that mutual exclusion constructs are not needed.

### 3.2.3 Fork-Join

The fork-join pattern solves the problem of needing to execute a set of computations in parallel. An instance of this pattern is given in Figure 3.10c. The computations can be different or identical and applied to different or identical data sets, an example of which was given in Figure 3.3 for the UAV example. The `par` statement forks the worker threads specified by the programmer. If there are no data dependencies between the worker threads, then shared variables are not needed.
shared int s=0,t=0;
void t1(void) {
    s=1;
    pause;
    int y=t;
}
void t2(void) {
    t=2;
    pause;
    int x=s;
}

(a) Point-to-point.

shared int s=0;

// Producer.
void t1(void) {
    s=1;
    pause;
    int y=t;
}

// Consumers.
void t2(void) {
    t=2;
    pause;
    int x=s;
    void t3(void) {
        pause;
        int y=s;
    }
}

(b) Broadcast.

void main(void) {
    par(wk1(),wk2());
} // Worker threads.
void wk1(void) {
    ...
}
void wk2(void) {
    ...
}

(c) Fork-join.

shared double result=0 combine all with total;
double total(double th1,double th2,double pre) {
    return (th1+th2);
}
void main(void) {
    par(pi(1,100), pi(101,200));
} // Worker thread.
void pi(int i,int j) {
    // Calculate pi series from term i to j
    result=piSeries(i,j);
}

(d) Map-reduce.

input int t=0;
shared int s1=0, s2=0, s3=0;
void main(void) {
    par(stage1(), par(stage2(),stage3()));
} void stage1(void) {
    while (1) {
        s1=process1(t);
        pause;
    }
}
void stage2(void) {
    pause;
    while (1) {
        s2=process2(s1);
        pause;
    }
}
void stage3(void) {
    pause; pause;
    while (1) {
        s3=process3(s2);
        pause;
    }
}

(e) Software pipeline.

typedef struct { char key; int value; } Map;
shared Map data[100] = { ... };
shared int found=-1 combine mod with left;
int left(int th1,int th2,int pre) { return th1; }
void main(void) {
    abort {
        par(search(3,0,49), search(3,50,99));
    } when (found != -1);
    printf("\%c\%d\",data[found].key,data[found].value);
}
void search(int value,int i,int j) {
    for (i; i <= j; i++) {
        if (data[i].value == value) { found=i; }
        pause;
    }
}

(f) Early-termination.

Figure 3.10: Instances of the design patterns in ForeC pseudo-code.
3.2 Design patterns

3.2.4 Map-Reduce

The map-reduce pattern solves the problem of needing to apply the same transform to each data element, followed by a reduction operation on the results. An instance of this pattern is given in Figure 3.10d. The map-reduce pattern is similar to the fork-join pattern, except the results from the worker threads are written to a common shared variable. Then, the results are reduced when the shared variable is resynchronized. Observe that this resynchronization is automatic in ForeC thanks to its shared variable semantics. In Figure 3.10d, the value of \( \pi \) is approximated by calculating part of an infinite series. Two threads are used to calculate the first 200 terms of the series. When the `par` terminates, the combine function reduces the results, giving an approximation of \( \pi \).

3.2.5 Software Pipeline

The software pipeline pattern solves the problem of needing to process a data stream in stages, typically needed in audio and video processing. An instance of this pattern is given in Figure 3.10e. The processing is broken down into different pipeline stages that work in parallel on different chunks of the data stream. The pattern forks a thread for each pipeline stage. Each stage gets a chunk of data, processes the data, and passes the result to the next stage. This is repeated until the data stream ends. Data is passed from each stage using the point-to-point pattern. The explicit use of buffers is not needed because threads always work on local copies of the shared variables. The pipeline is synchronous because the stages pause before processing their next chunk of data. Hence, the throughput is determined by the slowest pipeline stage. To initialize the pipeline correctly, each stage must wait for their initial chunk of data. The waiting is achieved by placing, at the start of the threads, a number `pause` statements equal to the number of preceding stages. The stages execute in parallel after all have waited for their initial chunk of data.

3.2.6 Early-Termination

The early-termination [259] pattern solves the problem of needing to terminate a search as soon as a match is found to avoid the need to explore the entire search space. An instance of this pattern is given in Figure 3.10f. The early-termination pattern is similar to the map-reduce pattern. In Figure 3.10f, we are searching a data set (defined on line 2), consisting of keys with associated values, for a key with the value 3. Two threads (line 8) search through both halves of the data set.
When a thread finds a match, its current position in the data set (always greater than -1) is assigned to the shared variable `found` (line 14). In this example, we are only interested in finding the first match, not all possible matches. Hence, when multiple matches are found, the combine function for `found` (line 4) only returns one of the matches. To preempt the search when a match is found, the threads are enclosed in an `abort` statement with the preemption condition “`found != -1`”. When a match is found, the preemption will trigger in the following tick due to the abort semantics.
3.3 Semantics

This section presents the semantics of ForeC as rewrite rules in the style of structural operational semantics (SOS) [260]. The semantics is inspired by that of other synchronous programming languages (Esterel [118] and PRET-C [50] in particular). The semantics is defined on a set of primitive ForeC constructs (the kernel of Table 3.4) from which the full ForeC constructs are derived. The kernel constructs are not used for compiling and only consider a subset of the C language: the assignment (=) and sequence (;) operators and the if and while statements. Table 3.5 shows how the ForeC constructs (Table 3.1) are translated into the kernel constructs (Table 3.4). This is exemplified by the translation of the ForeC constructs in Figure 3.11b into the kernel constructs in Figure 3.11c. The translations for input, output, and pause are straightforward. A shared variable is translated into a global variable and a copy kernel statement is placed at the start of every thread body in the scope of the shared variable. The copy kernel statement initiates the copying of the shared variables when the threads are forked. The copy kernel statement is also used to copy the shared variables at the start of each local tick. The par statement is translated by prefixing each thread body \( f \) with a unique identifier \( t \) to allow the semantics to distinguish the body of one thread from another. The par kernel statement handles the resynchronization of the shared variables. Traditionally, traps [118] are used to translate aborts and other complex preemption statements. In contrast, a simpler abort translation is possible in ForeC because abort is the only type of preemption statement. Each abort is assigned a unique identifier \( a \) and translated into the status and abort kernel statements. The status kernel statement is needed to define the immedi-

<table>
<thead>
<tr>
<th>Kernel Construct</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>Empty statement</td>
</tr>
<tr>
<td>( f; f )</td>
<td>Sequence operator</td>
</tr>
<tr>
<td>( \text{var} = \text{exp} )</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>while (( \text{exp} )) ( f )</td>
<td>Loop</td>
</tr>
<tr>
<td>if (( \text{exp} )) ( f ) else ( f )</td>
<td>Conditional</td>
</tr>
<tr>
<td>copy</td>
<td>Creates copies of shared variables</td>
</tr>
<tr>
<td>pause</td>
<td>Barrier synchronization</td>
</tr>
<tr>
<td>par(( t:f ), ( t:f ))</td>
<td>Fork/join parallelism</td>
</tr>
<tr>
<td>status(( a ), ( \text{exp} ))</td>
<td>Initial preemption status</td>
</tr>
<tr>
<td>weak? abort(( a ), ( f ))</td>
<td>Abort</td>
</tr>
</tbody>
</table>

Table 3.4: ForeC kernel constructs. \( f \) is an arbitrary composition of kernel constructs, \( \text{var} \) is a variable, \( \text{exp} \) is an expression, \( t \) is a thread identifier, and \( a \) is an abort identifier. A question mark means that the preceding symbol is optional.
### Table 3.5: Structural translations of the ForeC constructs (Table 3.1) to kernel constructs (Table 3.4).

<table>
<thead>
<tr>
<th>ForeC Construct</th>
<th>ForeC Kernel Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>input and output</td>
<td>Translated into global variables.</td>
</tr>
<tr>
<td>shared</td>
<td>Translated into global variables and the copy kernel statement is placed at start of every thread body.</td>
</tr>
<tr>
<td>pause</td>
<td>pause</td>
</tr>
<tr>
<td>par((f, f))</td>
<td>par((t:f, t:f))</td>
</tr>
<tr>
<td>weak? abort (f) when ((exp))</td>
<td>status((a, 0)); weak? abort((a, f))</td>
</tr>
<tr>
<td>weak? abort (f) when immediate ((exp))</td>
<td>status((a, exp)); weak? abort((a, f))</td>
</tr>
</tbody>
</table>

The abort behavior of an abort and it takes the unique identifier \(a\) and an expression. The expression is 0 (zero) for a non-immediate abort, but is \(exp\) (preemption condition) for an immediate abort. The abort kernel statement takes the unique identifier \(a\) and the abort body \(f\). The following section describes the assumptions on ForeC kernel programs to simplify the presentation of the formal semantics. The notations, semantic functions, and rewrite rules are then presented.

#### 3.3.1 Assumptions

We make the following assumptions about ForeC programs. 1) All programs follow safety-critical coding practices, as discussed in Chapters 1.5 and 3.1.7. For example, all expressions are side-effect free, dynamic memory allocation (e.g., malloc) is not used, and loops that can iterate an unbounded number of times without pausing are rejected. This assumption limits us to a deterministic subset of the C language. 2) All recursive function calls or forking of threads are bounded. This assumption prevents the unbounded execution of functions and threads, leading to unbounded memory use and execution time.

To simplify the presentation of the semantics, we assume that the following transformations have been performed on ForeC programs. 1) Inlining of functions at their call sites, so that the semantics can ignore function calls. 2) Renaming variables uniquely and hoisting their declarations up to the program’s global scope, so that the semantics can ignore memory allocation and focus on the semantics of private variables (accessible to only one thread) and shared variables. 3) Replacing pointers with the variables they reference, so that the semantics can ignore pointer analysis [72, 73]. Consider the program of Figure 3.11a that is transformed into the equivalent program of Figure 3.11b. The shared variable declaration for \(s\) (line 4 in Figure 3.11a) is hoisted to the global scope (line 3 in Figure 3.11b).
3.3 Semantics

(a) Original program.

(b) Transformed program.

(c) Translated kernel program.

Figure 3.11: Example of transforming and translating a ForeC program into the kernel constructs.

The function \( f \) (line 8 in Figure 3.11a) is inlined into the \texttt{abort} body (line 7 in Figure 3.11b) and the pointer inside \( f \) is replaced by the variable \( x \) it references.

3.3.2 Notation

The rewrite rules have the following form in the style of structural operational semantics (SOS) [260]:

\[
\langle S \rangle \ t : f \xrightarrow{k}{\langle S' \rangle} \ t : f'
\]

This notation describes a program fragment \( f \) belonging to thread \( t \), in the program state \( \langle S \rangle \) and with inputs \( I \), which reacts and modifies the program state to \( \langle S' \rangle \), emits the completion code \( k \), and becomes the new program fragment \( f' \). All the (globally declared) \texttt{input} variables are stored in \( I \). Let \( T \) be the set of all threads in the program. Let \( \langle S \rangle \) be defined as \( \langle E, A \rangle \), where:

- \( E \) is an environment that maps the program’s global scope to the global variables and the threads’ scopes to their local copies of shared variables. Specifically, \( E \) is a partial function that maps the global scope (denoted by \( G \)) and threads \( t \in T \) to a store \((\text{Store})\) of variables. Let \( \text{Id} = T \cup \{G\} \), then \( E : \text{Id} \leftrightarrow \text{Store} \). Since all variables are globally declared (Chapter 3.3.1), \( E[G] \) stores all the output, shared, and private variables. \( E[t] \) stores thread \( t \)'s copies of shared variables. The store \((\text{Store})\) is a partial
function that maps variables \((\text{var} \in \text{Var})\) to values \((\text{val} \in \text{Val})\) and statuses \((\text{sts} \in \text{Sts})\), \(\text{Store} : \text{Var} \rightarrow (\text{Val, Sts})\). Statuses are used to define the behavior of the combine policies and can be \(\text{pre}\) (previous resynchronized value), \(\text{mod}\) (modified value), \(\text{cmb}\) (partially combined value), or \(\text{pvt}\) (for a private variable). In \(E[\mathcal{G}]\), the status of a private variable is always \(\text{pvt}\) and the status of a shared variable is always \(\text{pre}\). In \(E[t]\), a thread’s copy of a shared variable always starts each local tick with the status \(\text{pre}\).

For example, \(E = \{\mathcal{G} \rightarrow \{\text{s} \rightarrow (1, \text{pre})\}, t1 \rightarrow \{\text{s} \rightarrow (3, \text{mod})\}\}\) for a program that has shared variable \(\text{s}\) with value 1 and thread \(t1\)’s modified copy of \(\text{s}\) with value 3. We use the notation \(E[t1][\text{s}]\) to look up the value and status \((3, \text{mod})\) of \(\text{s}\) in \(t1\)’s store. We use the notations \(E[t1][\text{s}].\text{val}\) and \(E[t1][\text{s}].\text{sts}\) to look up its value and status, respectively.

• \(A\) is a partial function that maps the \(\text{abort}\) identifiers \((a \in \mathcal{A})\) in the program to values \((\text{val} \in \text{Val})\) representing their preemption status, \(A : \mathcal{A} \rightarrow \text{Val}\). An \(\text{abort}\) with a non-zero value means that its preemption condition is \(\text{true}\) and has been triggered.

For example, \(A = \{a1 \rightarrow 1, a2 \rightarrow 0\}\) for a program that has \(\text{aborts a1}\) and \(a2\) with the statuses 1 and 0, respectively. We use the notation \(A[a1]\) to look up the status of \(\text{abort a1}\).

The transition of a program fragment from \(f\) to \(f'\) is encoded by the completion code \(k\), where:

\[
k = \begin{cases} 
0 & \text{If the transition terminates.} \\
1 & \text{If the transition pauses.} \\
\bot & \text{Otherwise (the transition continues).}
\end{cases}
\]
3.3 Semantic Functions

This section describes the semantic functions that are used by the rewrite rules to keep the ForeC semantics concise.

Statically Known Information

The following semantic functions return statically known information about the program:

- **GetParent(t):** Returns the parent of thread $t$. If $t = \text{main}$, then “main” is returned.
- **GetShared($G$):** Returns the set of all shared variables declared in the program.
- **GetShared($t$):** Returns the set of all shared variables that the body of thread $t$ accesses (reads or writes).
- **GetCombine($var$):** Returns the combine function of shared variable $var$.
- **GetPolicy($var$):** Returns the combine policy of shared variable $var$.
- **GetExp($a$):** Returns the preemption condition $exp$ of $\text{abort } a$.

Figure 3.12 exemplifies the use of these functions on the program in Figure 3.11c.

Eval

The semantic function $\text{Eval}(E, I, id, exp)$ follows the evaluation rules of C to evaluate the expression $exp$ and return its value. During the evaluation, a variable’s value is retrieved with the semantic function $\text{GetVal}(E, I, id, var)$ described by Algorithm 1. The inputs to the algorithm are: the program’s environment $E$, the inputs $I$, the identifier $id$ of the (global or thread) store to try and retrieve the value from, and the variable $var$ of interest. The output is a value $val$. If $var$ is an input, then line 2 returns its value. Otherwise, if $var$ is in $id$’s store, then line 4 returns its value. Otherwise, line 6 returns the global value of $var$.
Algorithm 1 \textsc{GetVal}(E, I, id, var): Gets the value of a given variable.

\begin{algorithm}
\caption{GetVal($E$, I, id, var): Gets the value of a given variable.}

\textbf{Input:} Program's environment $E$, inputs $I$, identifier $id$ of the store to search, and variable $var$ of interest.

\textbf{Output:} Value of $var$.

\begin{algorithmic}[1]
\begin{align*}
1: & \text{ if } var \in I \text{ then} & \text{\hspace{1em}} & \triangleright \text{ If } var \text{ is an input.} \\
2: & \quad \text{ return } I[var] & \text{\hspace{1em}} & \triangleright \text{ Return the input value of } var. \\
3: & \text{ else if } var \in E[id] \text{ then} & \text{\hspace{1em}} & \triangleright \text{ Otherwise, if a local copy of } var \text{ exists.} \\
4: & \quad \text{ return } E[id][var].val & \text{\hspace{1em}} & \triangleright \text{ Return the value of } var \text{ from } id\text{'s store.} \\
5: & \text{ else } \text{ end if} \\
6: & \quad \text{ return } E[G][var].val & \text{\hspace{1em}} & \triangleright \text{ Otherwise, return the global value of } var.
\end{align*}
\end{algorithmic}
\end{algorithm}

\textbf{COPY}

The semantic function \textsc{Copy}($E$, $t$) creates the copies of shared variables that thread $t$ needs but does not already have. That is, if thread $t$ already has a copy of the shared variable $var$, then \textsc{Copy} skips the copying of $var$. This ensures that, when a \texttt{par} terminates, the combined values assigned to the parent thread will not be overwritten. Also, when a \texttt{par} forks and joins in the same tick, the parent thread's existing copies will not be overwritten. The \textsc{Copy} function is described by Algorithm 2. The inputs to the algorithm are: the program's environment $E$ and a thread $t$. The output is an updated environment $E$. Line 1 considers each shared variable that is accessed in the thread's body. For each shared variable, line 2 checks if a copy already exists. If it does not exist, then lines 4–5 copies the the parent thread's copy if available, otherwise from the shared variable (line 7). Line 11 returns the updated environment $E$.

\textbf{COMBINE}

The semantic function \textsc{Combine}($E$, $t_1$, $t_2$, $t_0$) combines all the copies of shared variables from two threads and is described by Algorithm 3. The inputs to the algorithm are: the program's environment $E$, two threads $t_1$ and $t_2$ to combine, and thread $t_0$ to store the combined values. The output is an updated environment $E$. Line 1 considers each shared variable\footnote{Recall from Chapter 3.3.2 that $E$ maps the global and thread scopes to their own store of variables, $E : Id \leftrightarrow Store$. Variables are mapped to a value and status, $Store : Var \leftrightarrow (Val, Sts)$ where $Sts = \{pre, mod, cmb, pvt\}$. A private variable has the status $pvt$, a shared variable has the status $pre$, and a thread’s copy of a shared variable starts each local tick with the status $pre$. The notation $E[t][var]$ looks up the value and status ($val, sts$) of thread $t$’s copy of $var$.} $var$. Line 2 gets the shared variable’s $\texttt{pre}$ value ($\texttt{preVal}$). For the combine policy \texttt{all}, the copies from both threads are combined if they exist. Thus, line 3 gets the set of threads $T$ that have a copy

Algorithm 2 \textsc{Copy}(E, t): Copies all the shared variables needed by a thread.

\textbf{Input:} Program’s environment $E$, and thread $t$.
\textbf{Output:} Updated environment $E$.

1: \textbf{for all} $\text{var} \in \text{GetShared}(t)$ \textbf{do} \quad \triangleright \text{For all shared variables needed by thread } t.
2: \quad \textbf{if} $\text{var} \notin E[t]$ \textbf{then} \quad \triangleright \text{If thread } t \text{ does not have a copy.}
3: \quad \quad \textbf{if} $\text{var} \in E[\text{GetParent}(t)]$ \textbf{then} \quad \triangleright \text{If its parent has a copy.}
4: \quad \quad \quad \text{val} := E[\text{GetParent}(t)][\text{var}].\text{val} \quad \triangleright \text{Value of its parent’s copy.}
5: \quad \quad \text{\text{E}[t][\text{var} \leftarrow (\text{val}, \text{pre})]} \quad \triangleright \text{Copy its parent’s copy.}
6: \quad \quad \text{else}
7: \quad \quad \quad \text{E}[t][\text{var} \leftarrow E[\mathcal{G}][\text{var}]} \quad \triangleright \text{Otherwise, copy the shared variable’s pre.}
8: \quad \quad \textbf{end if}
9: \quad \textbf{end if}
10: \textbf{end for}
11: \textbf{return } E

Algorithm 3 \textsc{Combine}(E, t_1, t_2, t_0): Combines the copies of shared variables from two threads.

\textbf{Input:} Program’s environment $E$, threads $t_1$ and $t_2$ to combine, and thread $t_0$ to store the combined values.
\textbf{Output:} Updated environment $E$.

1: \textbf{for all} $\text{var} \in \text{GetShared}(\mathcal{G})$ \textbf{do} \quad \triangleright \text{For all shared variables.}
2: \quad \quad \text{\text{preVal} := E[\mathcal{G}][\text{var}].\text{val}} \quad \triangleright \text{Get the pre of var.}
3: \quad \quad \text{T := \{t | t \in \{t_1, t_2\}, \text{var} \in E[t]\}} \quad \triangleright \text{Set of threads with a copy of var.}
4: \quad \text{\textbf{if } \text{GetPolicy}(\text{var}) = \text{new} \textbf{then}}
5: \quad \quad \text{// Keep only the copies that differ from \text{preVal} or are partially combined.}
6: \quad \quad \text{T := \{t | t \in T, E[t][\text{var}].\text{val} \neq \text{preVal} \lor E[t][\text{var}].\text{sts} = \text{cmb}\}}
7: \quad \text{\textbf{else if } \text{GetPolicy}(\text{var}) = \text{mod} \textbf{then}}
8: \quad \quad \text{// Keep only the modified or partially combined copies.}
9: \quad \quad \text{T := \{t | t \in T, E[t][\text{var}].\text{sts} \in \{\text{mod, cmb}\}\}}
10: \quad \textbf{end if}
11: \quad \quad \textbf{if } |T| = 2 \textbf{then} \quad \triangleright \text{If there are two copies to combine.}
12: \quad \quad \quad \text{c := \text{GetCombine(\text{var})}} \quad \triangleright \text{Get the combine function of \text{var}.}
13: \quad \quad \quad \text{val := c(E[t_1][\text{var}].\text{val}, E[t_2][\text{var}].\text{val}, \text{preVal})} \quad \triangleright \text{Combine the copies.}
14: \quad \quad \quad \text{E[t_0][\text{var} \leftarrow (\text{val}, \text{cmb})]} \quad \triangleright \text{Assign the combined value to } t_0.
15: \quad \quad \textbf{else if } |T| = 1 \textbf{then} \quad \triangleright \text{Otherwise, there is only one copy.}
16: \quad \quad \quad \text{E[t_0][\text{var} \leftarrow (E[t \in T][\text{var}].\text{val}, \text{cmb})]} \quad \triangleright \text{Assign the only copy to } t_0.
17: \quad \quad \textbf{end if}
18: \quad \textbf{end for}
19: \textbf{return } E \setminus \{t_1, t_2\}
of the shared variable. If the combine policy is \texttt{new}, then line 6 keeps only the copies with values that differ from the shared variable’s \texttt{pre} value \((E[t][\texttt{var}].\texttt{val} \neq \texttt{preVal})\) or are partially combined copies \((E[t][\texttt{var}].\texttt{sts} = \texttt{cmb})\). If the combine policy is \texttt{mod}, then line 9 keeps only the modified or partially combined copies \((E[t][\texttt{var}].\texttt{sts} \in \{\texttt{mod, cmb}\})\). If two copies are found, then line 13 gets the shared variable’s combine function \((c)\) and line 14 computes the combined value. Line 15 assigns the combined value to thread \(t_0\) with the status \texttt{cmb} because it is a partially combined value. If only one copy is found, then line 17 assigns the value of that copy to thread \(t_0\) with the status \texttt{cmb}. Line 20 returns the updated environment \(E\) without thread \(t_1\) and \(t_2\)’s store.

### 3.3.4 The Structural Operational Semantics

This section presents the operational semantics of the kernel constructs presented in Table 3.4.

**The \texttt{nop} Statement**

The \texttt{nop} statement does nothing and terminates instantly:

\[
(E, A) \ t : \texttt{nop} \xrightarrow{\sigma_t} (E, A) \ t :
\]

(nop)

**The \texttt{copy} Statement**

The \texttt{copy} statement copies the shared variables needed by thread \(t\) and terminates instantly. The combining of the copies is handled by the \texttt{par} statement:

\[
(E, A) \ t : \texttt{copy} \xrightarrow{\sigma_t} (\text{Copy}(E, t), A) \ t :
\]

(copy)

**The \texttt{pause} Statement**

The \texttt{pause} statement rewrites into the \texttt{copy} statement and pauses. The \texttt{copy} statement ensures that thread \(t\) starts its next local tick by copying the shared variables it needs (the \texttt{pre} values are copied):

\[
(E, A) \ t : \texttt{pause} \xrightarrow{\sigma_t} (E, A) \ t : \texttt{copy}
\]

(pause)
The status Statement

The status statement sets \texttt{abort} \(a\)'s preemption status to the value of the expression \(exp\), and then it terminates instantly:

\[
\langle E, A \rangle t : \text{status}(a, exp) \xrightarrow{0}{1} \langle E, A[a \leftarrow \text{EVAL}(E, I, t, exp)] \rangle t : \text{(status)}
\]

The abort Statement

The abort of \(a\) executes its body \(f\) if its preemption has not been triggered:

\[
\begin{align*}
(E, A) \ t : f & \xrightarrow{k \in (1.\ldots)} (E', A') \ t : f' & (A[a] = 0) & (\text{abort-1}) \\
(E, A) \ t : \text{weak abort}(a, f) & \xrightarrow{k}{1} (E', A') \ t : \text{weak abort}(a, f')
\end{align*}
\]

The abort terminates normally if its body terminates and its preemption has not been triggered:

\[
(E, A) \ t : f \xrightarrow{0}{1} (E', A') \ t :
\]

\[
(E, A) \ t : \text{weak abort}(a, f) \xrightarrow{0}{1} (E', A') \ t : (A[a] = 0) & (\text{abort-2})
\]

The weak abort terminates normally if its body terminates, even if its preemption has been triggered:

\[
(E, A) \ t : f \xrightarrow{0}{1} (E', A') \ t :
\]

\[
(E, A) \ t : \text{weak abort}(a, f) \xrightarrow{0}{1} (E', A') \ t : (A[a] \neq 0) & (\text{abort-3})
\]

The weak abort allows its body to execute until it pauses, even if its preemption has been triggered:

\[
(E, A) \ t : f \xrightarrow{0}{1} (E', A') \ t : f'
\]

\[
(E, A) \ t : \text{weak abort}(a, f) \xrightarrow{0}{1} (E', A') \ t : \text{weak abort}(a, f') (A[a] \neq 0) & (\text{abort-4})
\]

The weak abort terminates if its body pauses and its preemption has been triggered, and then it rewrites into the copy statement because it could be the start of thread \(t\)'s local tick\(^2\):

\[
(E, A) \ t : f \xrightarrow{1}{1} (E', A') \ t : f'
\]

\[
(E, A) \ t : \text{weak abort}(a, f) \xrightarrow{1}{1} (E', A') \ t : \text{copy} (A[a] \neq 0) & (\text{abort-5})
\]

---

\(^2\) The abort body could have executed a par statement over several ticks. In this case, when the par is preempted, thread \(t\) will start its local tick.
The strong `abort` terminates without executing its body if its preemption has been triggered, and then it rewrites into the `copy` statement because it could be the start of thread `t`’s local tick:

\[
A[a] \neq 0 \\
\langle E, A \rangle t : \text{abort}(a, f) \xrightarrow{\frac{1}{t}} \langle E, A \rangle t : \text{copy}
\]

The Assignment Operator (\(=\))

The assignment operator evaluates the expression `exp` into a value `val = \text{EVAL}(E, I, t, exp)`. If `var` is a shared variable (assign-shared), then the value `val` and status `mod` is assigned to the thread’s copy in `E[t]`. Otherwise, if `var` is a private variable (assign-private), then the value `val` and status `pvt` is assigned to the global variable in `E[G]`:

\[
\begin{align*}
\text{var} \in \text{GetShared}(t) \\
\langle E, A \rangle t : \text{var} = \text{exp} \xrightarrow{0} \langle E[t][\text{var}(\text{val}, \text{mod})], A \rangle t : \\
\text{var} \notin \text{GetShared}(t) \\
\langle E, A \rangle t : \text{var} = \text{exp} \xrightarrow{0} \langle E[G][\text{var}(\text{val}, \text{pvt})], A \rangle t : 
\end{align*}
\]

The if–else Statement

A conditional construct is rewritten into one of its branches, depending on the value of its condition `exp`:

\[
\begin{align*}
\text{eval}(E, I, t, exp) \neq 0 \\
\langle E, A \rangle t : \text{if} \ (exp) f_1 \ \text{else} \ f_2 \xrightarrow{\frac{1}{t}} \langle E, A \rangle t : f_1
\end{align*}
\]

\[
\begin{align*}
\text{eval}(E, I, t, exp) = 0 \\
\langle E, A \rangle t : \text{if} \ (exp) f_1 \ \text{else} \ f_2 \xrightarrow{\frac{1}{t}} \langle E, A \rangle t : f_2
\end{align*}
\]

---

3 In addition to footnote 2, when thread `t` starts its local tick, the preemption of the strong `abort` prevents `abort`’s body from executing a `copy` statement.

4 Recall from Chapter 3.3.2 that `E` maps the global and thread scopes to their own store of variables, \(E : Id \mapsto \text{Store}\). Variables are mapped to a value and status, \(\text{Store} : \text{Var} \mapsto (\text{Val}, \text{Sts})\) where \(\text{Sts} = \{\text{pre}, \text{mod}, \text{cmb}, \text{pvt}\}\). A private variable has the status `pvt`, a shared variable has the status `pre`, and a thread’s copy of a shared variable starts each local tick with the status `pre`. The notation `E[t][var]` looks up the value and status `(val, sts)` of thread `t`’s copy of `var`. 
The while Statement

The body of a loop statement is unrolled once, depending on the value of its condition \( \text{exp} \):

\[
\text{Eval}(E, I, t, \text{exp}) \neq 0 \\
\langle E, A \rangle \ t: \text{while} (\text{exp}) \ f \xrightarrow{k \in \{1, \ldots, k\}} \langle E, A' \rangle \ t: f' \\
\langle E, A \rangle \ t: \text{while} (\text{exp}) \ f \xrightarrow{0} \langle E, A \rangle \ t:
\]

(loop-then)

\[
\text{Eval}(E, I, t, \text{exp}) = 0 \\
\langle E, A \rangle \ t: \text{while} (\text{exp}) \ f \xrightarrow{0} \langle E, A \rangle \ t:
\]

(loop-else)

The Sequence Operator (;)

For a sequence of program fragments, the first fragment \( f_1 \) must terminate before the second fragment \( f_2 \) can be rewritten. In other words, the (seq-left) rule applies up to the micro-step during which \( f_1 \) emits the completion code 0. At this point, the (seq-right) rule applies. The (seq-left) rule emits the completion code of the first fragment:

\[
\langle E, A \rangle \ t: f_1 \xrightarrow{k \in \{1, \ldots, k\}} \langle E', A' \rangle \ t: f_1' \\
\langle E, A \rangle \ t: f_1; f_2 \xrightarrow{k \in \{1, \ldots, k\}} \langle E', A' \rangle \ t: f_1'; f_2
\]

(seq-left)

\[
\langle E, A \rangle \ t: f_1 \xrightarrow{0} \langle E', A' \rangle \ t:
\]

(seq-right)

The par Statement

The par statement allows both its child threads, \( t_1 \) and \( t_2 \), to execute in parallel. The parent thread is \( t_0 \):

\[
\langle E, A \rangle \ t_1: f_1 \xrightarrow{1} \langle E', A' \rangle \ t_1: f_1' \langle E, A \rangle \ t_2: f_2 \xrightarrow{1} \langle E'', A'' \rangle \ t_2: f_2'
\]

(par-1)

\[
\langle E, A \rangle \ t_0: \text{par}(t_1: f_1, t_2: f_2) \xrightarrow{1} \langle E^A, A^A \rangle \ t_0: \text{par}(t_1: f_1', t_2: f_2')
\]

The variables that changed in \( E' \) or \( E'' \) are aggregated to form \( E^A \) by taking the union of the changes in \( E' \) (i.e., \( E' \setminus (E' \cap E) \)) and in \( E'' \) (i.e., \( E'' \setminus (E'' \cap E) \)) with the remaining unchanged variables (i.e., \( E' \cap E'' \)). Note that intersecting two environments, e.g., \( E' \cap E'' \), produces a new environment containing the variables that have the same values and statuses in \( E' \) and \( E'' \). Also, parallel threads always modify the same environment \( E \) in a mutually exclusive manner. Indeed, the (assign-shared) rule only allows a thread to access its own copies of shared variables and the (assign-private) rule only allows a thread to access its own private variables. Thus, \( E^A = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'') \). Similarly,
the preemption statuses that changed in $A'$ and $A''$ are aggregated to form $A^A = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'')$.

If a child thread can complete its local tick, by pausing or terminating, then it will wait for its sibling to complete their local tick. The waiting is captured by stopping the child thread from taking its transition:

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{k \in (0,1]}{I}} \langle E', A' \rangle \ t_1 : f'_1 \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle E'', A'' \rangle \ t_0 : \text{par}(t_1 : f'_1, t_2 : f'_2)
\end{array}
$$

(par-2)

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{1}{I}} \langle E', A' \rangle \ t_1 : f'_1 \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle E'', A'' \rangle \ t_0 : \text{par}(t_1 : f'_1, t_2 : f'_2)
\end{array}
$$

(par-3)

The `par` pauses if both its child threads pause. The changes made to $E$ and $A$ are aggregated into $E^A$ and $A^A$, respectively, as defined for the (par-1) rule. The copies of shared variables from the child threads are combined and assigned to their parent thread, thanks to the semantic function `COMBINE`:

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{1}{I}} \langle E', A' \rangle \ t_1 : f'_1 \\
\langle E, A \rangle \ t_2 : f_2 \xrightarrow{\frac{1}{I}} \langle E'', A'' \rangle \ t_2 : f'_2 \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle \ t_0 : \text{par}(t_1 : f'_1, t_2 : f'_2)
\end{array}
$$

(par-4)

Otherwise, the `par` terminates if both its child threads terminate. The `par` rewrites into the `copy` statement because it could be the start of parent $t_0$’s local tick\(^5\):

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{0}{I}} \langle E', A' \rangle \ t_1 : \\
\langle E, A \rangle \ t_2 : f_2 \xrightarrow{\frac{0}{I}} \langle E'', A'' \rangle \ t_2 : \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle \ t_0 : \text{copy}
\end{array}
$$

(par-5)

If only one child thread terminates while the other pauses, then the terminated child thread rewrites into the `nop` statement and the `par` pauses:

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{0}{I}} \langle E', A' \rangle \ t_1 : \\
\langle E, A \rangle \ t_2 : f_2 \xrightarrow{\frac{1}{I}} \langle E'', A'' \rangle \ t_2 : f'_2 \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle \ t_0 : \text{par}(t_1 : \text{nop}, t_2 : f'_2)
\end{array}
$$

(par-6)

$$
\begin{array}{c}
\langle E, A \rangle \ t_1 : f_1 \xrightarrow{\frac{1}{I}} \langle E', A' \rangle \ t_1 : f'_1 \\
\langle E, A \rangle \ t_2 : f_2 \xrightarrow{\frac{0}{I}} \langle E'', A'' \rangle \ t_2 : \\
\langle E, A \rangle \ t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\frac{1}{I}} \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle \ t_0 : \text{par}(t_1 : f'_1, t_2 : \text{nop})
\end{array}
$$

(par-7)

\(^5\) The `par` statement could have executed over several ticks. In this case, when the `par` terminates, the parent thread $t_0$ will start its local tick.
Tick Completion

A tick completes if the \texttt{main} thread pauses or terminates. If the \texttt{main} thread is executing a \texttt{par} statement, then a tick completes when all its child threads and nested child threads have paused or terminated. The shared variables are resynchronized, the preemption statuses are reevaluated, the outputs are emitted, and the inputs are sampled:

\[
\frac{\langle E, A \rangle \; \text{main} : f \; \mathcal{I} \; \mathcal{E} \rightarrow \langle E', A' \rangle \; \text{main} : f'}{\langle E, A \rangle \; \text{main} : f \; \mathcal{I} \; \mathcal{E} \rightarrow \langle E'', A'' \rangle \; \text{main} : f'} \quad \text{(tick)}
\]

The rules for the \texttt{par} statement ensures that, when the tick ends, \texttt{main}'s store \( E'[\text{main}] \) has the combined values from all its child threads. The shared variables\(^6\) are resynchronized by assigning the combined values from \texttt{main}'s store \( E'[\text{main}] \) to their corresponding shared variables in the global store \( E'[\mathcal{G}] \). Since \texttt{main}'s store is no longer needed, it is removed from the program's environment. Thus, for all \texttt{var} in \( E'[\text{main}] \), we have \( E'' = E'[\mathcal{G}][\text{var} \leftarrow (E'[\text{main}][\text{var}.\text{val}.\text{pre}]) \setminus \{\text{main}\} \). All the preemption statuses are updated by evaluating their preemption conditions with the resynchronized shared variables in \( E''[\mathcal{G}] \). Thus, for all \texttt{a} in \( A' \), we have \( A'' = A'[\text{a} \leftarrow \text{Eval}(E'', I, \mathcal{G}, \text{GetExp}(a))] \).

---

\(^6\) Recall from Chapter 3.3.2 that \( E \) maps the global and thread scopes to their own store of variables, \( E : Id \leftrightarrow \text{Store} \). Variables are mapped to a value and status, \( \text{Store} : \text{Var} \leftrightarrow (\text{Val}, \text{Sts}) \). In \( E[\mathcal{G}] \), shared variables have the status \text{pre}. The notation \( E[t][\text{var}] \) looks up the value and status \( (\text{val}, \text{sts}) \) of thread \( t \)'s copy of \texttt{var}.\]
3.4 Illustrations

This section provides two examples of how ForeC programs execute. The executions are given as sequences of rewrites.

3.4.1 Program One

The first program illustrates parallel execution using the \texttt{par} statement. Figure 3.13a presents the ForeC program and Figure 3.13c illustrates the program’s control-flow. In Figure 3.13c, the triangle represents the forking of threads while the inverted triangle represents the joining of threads.

In the program’s first tick, the parent thread \texttt{main} begins its local tick by forking two child threads, \texttt{t1} and \texttt{t2}. The child threads start their local ticks by copying the shared variable \texttt{s}. Thread \texttt{t1} pauses while thread \texttt{t2} assigns 4 to its copy of \texttt{s} and terminates. The first tick ends and the shared variable \texttt{s} is resynchronized. Using the combine policy \texttt{all}, the \texttt{pre} of \texttt{s} becomes \texttt{plus}(0, 4, 0) = 4. In the program’s second tick, thread \texttt{t1} starts its local tick by creating a copy of \texttt{s}, assigning 3 to its copy of \texttt{s}, and then terminating. The \texttt{par} terminates because threads \texttt{t1} and \texttt{t2} have now joined. Because only thread \texttt{t1} has a copy of \texttt{s}, that copy is assigned directly to its parent thread \texttt{main}. The \texttt{main} thread starts its local tick which results in the program terminating. The second tick ends and the shared variable \texttt{s} is resynchronized. The \texttt{pre} of \texttt{s} is 3 because only the \texttt{main} thread has a copy of \texttt{s}.

Before we apply the rewrite rules to the program, it is structurally translated into Figure 3.13b (see the start of Chapter 3.3). Note that the semantic functions \texttt{GetShared(t1)}, \texttt{GetShared(t2)} and \texttt{GetShared(G)} all return \{\texttt{s}\}, and
GetShared(main) returns \emptyset. Initially, the set of preemption statuses \( A \) is \emptyset. The program’s environment \( E \) and its derivatives are defined in Figure 3.14.

**Step 1:** Start the tick by applying the (seq-right) and (copy) rules. The copy statement has no effect because thread \( \text{main} \) does not access any shared variables.

\[
\begin{align*}
    \text{(copy)} & \quad \frac{\langle E, A \rangle \text{main:copy} \xrightarrow{0} \langle E, A \rangle \text{main:}}{
        \langle E, A \rangle \text{main:copy;\par(t1:\{copy; pause;s=3\};t2:\{copy;s=4\})} \xrightarrow{\text{seq-right}} \langle E, A \rangle \text{main:par(t1:\{copy; pause;s=3\};t2:\{copy;s=4\})}
    }
\end{align*}
\]

**Step 2:** Apply the (par-1) rule. Additionally, apply the (seq-right) and (copy) rules to both threads. The environments of both threads, \( E^1 \) and \( E^2 \), are aggregated into \( E^3 \).

\[
\begin{align*}
    \text{(copy)} & \quad \frac{\langle E, A \rangle \text{t1:copy} \xrightarrow{0} \langle E^1, A \rangle \text{t1:}}{
        \langle E, A \rangle \text{t1:copy; \text{pause;s=3}} \xrightarrow{\text{seq-right}} \langle E^1, A \rangle \text{t1:pause;s=3}
    }
    \quad \frac{\langle E, A \rangle \text{t2:copy} \xrightarrow{0} \langle E^2, A \rangle \text{t2:}}{
        \langle E, A \rangle \text{t2:copy; \text{pause;s=4;}} \xrightarrow{\text{par-1}} \langle E^2, A \rangle \text{t2:copy; \text{pause;s=4;}}
    }
\end{align*}
\]

**Step 3:** Apply the (tick) and (par-7) rules. Additionally, apply the (seq-left) and (pause) rules to the first thread and the (assign-shared) rule to the second thread.

**The program completes the tick.** Note that when the (par-7) rule is applied, the aggregated environment is the same as \( E^4 \), which is then combined to be \( E^5 \). When the (tick) rule is applied, \( E^5 \) is resynchronized to be \( E^6 \).
Step 4: Start the next tick by applying the (par-3) rule. Additionally, apply the (seq-right) and (copy) rules to the first thread and the (nop) rule to the second thread.

Step 5: Apply the (par-5) rule. Additionally, apply the (assign-shared) rule to the first thread and the (nop) rule to the second thread. Note that when the (par-5) rule is applied, the aggregated environment is the same as $E^8$, which is then combined to be $E^9$.

Step 6: Apply the (tick) and (copy) rules. The environment $E^9$ is resynchronized to be $E^{10}$. The tick ends and the program terminates.
3.4 Illustrations

3.4.2 Program Two

The second program illustrates preemption using an immediate and weak abort statement. Figure 3.15a presents the ForeC program and Figure 3.15c illustrates the program’s control-flow. In Figure 3.15c, the pair of decorated diamonds represents the scope of the abort body.

In the program’s first tick, the main thread reaches the immediate and weak abort and immediately evaluates the preemption condition (x==1). The condition evaluates to true and the preemption is triggered. Since the abort is weak, the preemption is taken only when execution reaches the pause, after the variable x has been incremented by 1. The abort terminates and, as a result, the main thread terminates. The first tick ends.

Before we apply the rewrite rules to the program, it is structurally translated into Figure 3.15b (see the start of Chapter 3.3). The copy kernel statement is not inserted into the program because shared variables are not used. Note that the semantic functions GET_SHARED(main) and GET_SHARED(G) all return Ø. The program’s environment E, preemption statuses A and their derivatives are defined in Figure 3.16.

**Step 1:** Start the tick by applying the (seq-right) and (status) rules. Note that
the abort’s preemption is triggered because the condition \(x==1\) evaluates to 1.

\[
\begin{align*}
\text{(status)} & \quad \langle E, A \rangle \text{ main: status}(a1,x==1) \xrightarrow{0} \langle E, A^1 \rangle \text{ main:} \\
\text{(seq-right)} & \quad \langle E, A \rangle \text{ main}: \text{status}(a1,x==1); \xrightarrow{1} \langle E, A^1 \rangle \text{ main: weak abort} \\
& \quad \text{weak abort}(a1,\{x++;\text{pause;}\}) \xrightarrow{1} \langle a1,\{x++;\text{pause;}\} \rangle
\end{align*}
\]

**Step 2:** Apply the (abort-4), (seq-right), and (assign-private) rules.

\[
\begin{align*}
\text{(assign-private)} & \quad \langle E, A^1 \rangle \text{ main: } x \notin \emptyset \\
\text{(seq-right)} & \quad \langle E, A^1 \rangle \text{ main: } x++; \xrightarrow{0} \langle E^1, A^1 \rangle \text{ main:} \\
\text{(abort-4)} & \quad \langle E, A^1 \rangle \text{ main: weak abort} \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: weak abort} \\
& \quad \langle a1,\{x++;\text{pause;}\} \rangle \xrightarrow{1} \langle a1,\{\text{pause;}\} \rangle
\end{align*}
\]

**Step 3:** Apply the (abort-5) and (pause) rules. Note that the preemption is taken because the abort’s body has reached a pause.

\[
\begin{align*}
\text{(pause)} & \quad \langle E^1, A^1 \rangle \text{ main: pause} \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: copy} \\
\text{(abort-5)} & \quad \langle E^1, A^1 \rangle \text{ main: weak abort}(a1,\{\text{pause;}\}) \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: copy} \\
& \quad (A^1[a1] \neq 0)
\end{align*}
\]

**Step 4:** Apply the (tick) and (copy) rules. The preemption statuses in \(A^1\) are updated to be \(A^2\). **The tick ends and the program terminates.**

\[
\begin{align*}
\text{(copy)} & \quad \langle E^1, A^1 \rangle \text{ main: copy} \xrightarrow{0} \langle E^1, A^1 \rangle \text{ main:} \\
\text{(tick)} & \quad \langle E^1, A^1 \rangle \text{ main: copy} \xrightarrow{0} \langle E^1, A^2 \rangle \text{ main:}
\end{align*}
\]
3.5 Definitions and Proofs

The semantics of the ForeC kernel constructs can be used to formally prove two desirable properties of safety-critical programs, called reactivity and determinism [253, 254]. A program is reactive if it always responds to changes in the environment, i.e., does not deadlock and produces outputs instantaneously. A program is deterministic if there is a unique proof for its response to the environment. The definitions for reactivity and determinism are normally based on a program’s tick, which is a sequence of transitions. Because the state of a ForeC program depends on the valuations of its variables, we define a stronger notion of reactivity and determinism based on program transitions.

**Definition 3.** A program \( t : f \) is **reactive** if, in any state \( S \), for any input configuration \( I \), there exists at least one transition (i.e., the program never deadlocks):

\[
\forall S, I : \exists S', f', k \text{ such that } \langle S \rangle t : f \xrightarrow{k}{I} \langle S' \rangle t : f'
\]

**Theorem 1.** All ForeC programs are reactive.

**Proof.** The proof can be shown by structural induction on \( t : f \).

**Base cases:** The (nop), (copy), (pause), (status), (assign-shared), (assign-private), (if-then), (if-else), (loop-then), and (loop-else) rules imply that the following kernel constructs have at least one transition:

\[
\langle S \rangle t : \text{nop} \xrightarrow{0}{I} \langle S \rangle t : \\
\langle S \rangle t : \text{copy} \xrightarrow{0}{I} \langle S' \rangle t : \\
\langle S \rangle t : \text{pause} \xrightarrow{1}{I} \langle S \rangle t : \text{copy} \\
\langle S \rangle t : \text{status}(a, exp) \xrightarrow{0}{I} \langle S' \rangle t : \\
\langle S \rangle t : \text{var}=exp \xrightarrow{0}{I} \langle S' \rangle t :
\]

\[
\langle S \rangle t : \text{if}(exp) f_1 \text{ else } f_2 \xrightarrow{1}{I} \langle S \rangle t : f_1 \text{ or } \langle S \rangle t : \text{if}(exp) f_1 \text{ else } f_2 \xrightarrow{1}{I} \langle S \rangle t : f_2 \\
\langle S \rangle t : \text{while}(exp) f \xrightarrow{1}{I} \langle S \rangle t : f; \text{ while}(exp) f \text{ or } \langle S \rangle t : \text{while}(exp) f \xrightarrow{0}{I} \langle S \rangle t : \\
\]

**Induction step:** The sequence operator (\( ; \)), **abort**, and **par** kernel statements are now of interest because they allow the composition of kernel constructs. For some \( t_1 : f_1 \) and \( t_2 : f_2 \) that are arbitrary compositions of kernel constructs, assume the induction hypotheses that they each have at least one transition:

\[
\exists S'_1, S'_2, f'_1, f'_2, k_1, k_2 \text{ such that } \langle S_1 \rangle t_1 : f_1 \xrightarrow{k_1}{I} \langle S'_1 \rangle t_1 : f'_1 \quad (H1) \\
\langle S_2 \rangle t_2 : f_2 \xrightarrow{k_2}{I} \langle S'_2 \rangle t_2 : f'_2 \quad (H2)
\]

Next, we show that the remaining sequence operator (\( ; \)), **abort**, and **par** kernel
The ForeC Language

statements have at least one transition.

1. Consider, \(t_1 : f_1; f_2\). Due to the induction hypotheses, the table below shows that at least one sequence rule can be applied to all possible completion codes \(k_1\) of the first program fragment \(f_1\). Note that the sequence rules do not consider the completion code \(k_2\) of the second program fragment \(f_2\):

\[
\begin{array}{c|c|c}
  & k_1 & \bot \\
\hline
0 & (seq-right) & 1 & (seq-left) \\
\end{array}
\]

That is, if \(k_1 = 0\) and the premise is true by the induction hypothesis (H1), then from the (seq-right) rule we have:

\[
\frac{(S_1) \ t_1 : f_1 \xrightarrow{k_1=0} S_1'} {S_1'}
\]

and if \(k_1 \in \{1, \bot\}\) and the premise is true by the induction hypothesis (H1), then from the (seq-left) rule we have:

\[
\frac{(S_1) \ t_1 : f_1 \xrightarrow{k_1} S_1'} {S_1'}
\]

Thus, any sequential composition of reactive programs has at least one transition and is, therefore, reactive.

2. Consider, \(t_1 : \text{weak? abort}(a_1, f_1)\). Due to the induction hypotheses, the table below shows that at least one abort rule can be applied to every combination of \(k_1\) and preemption status \(A[a_1]\):

\[
\begin{array}{c|c|c|c|c}
  & \text{Strong abort, } k_1 & \text{Weak abort, } k_1 \\
\hline
A[a_1] = 0 & (abort-2) & (abort-3) & (abort-2) & (abort-3) \\
A[a_1] \neq 0 & (abort-6) & (abort-3) & (abort-5) & (abort-4) \\
\end{array}
\]

For example, if \(k_1 = 0\) and \(A[a_1] = 0\) and the premise is true by the induction hypothesis (H1), then from the (abort-2) rule we have:

\[
\frac{(S_1) \ t_1 : f_1 \xrightarrow{k_1=0} S_1'} {S_1'}
\]
Thus, any preemptive composition of reactive programs has at least one transition and is, therefore, reactive.

3. Consider, $t : \text{par}(t_1 : f_1, t_2 : f_2)$. Due to the induction hypotheses, the table below shows that at least one $\text{par}$ rule can be applied to every combination of $k_1$ and $k_2$:

<table>
<thead>
<tr>
<th>$k_1$</th>
<th>$k_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(par-5) (par-6) (par-2)</td>
</tr>
<tr>
<td>1</td>
<td>(par-7) (par-4)</td>
</tr>
<tr>
<td>⊥</td>
<td>(par-3) (par-1)</td>
</tr>
</tbody>
</table>

For example, if $k_1 = 0$ and $k_2 = 0$ and the premise is true by the induction hypotheses (H1) and (H2), then from the (par-5) rule we have:

$$(\text{par-5}) \frac{\langle S \rangle t_1 : f_1 \xrightarrow{k_1=0} \langle S'_1 \rangle t_1 : \langle S \rangle t_2 : f_2 \xrightarrow{k_2=0} \langle S'_2 \rangle t_2 :}{\langle S \rangle t : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\bot} \langle S'' \rangle t : \text{copy}}$$

Thus, any parallel composition of reactive programs has at least one transition and is, therefore, reactive.

**Definition 4.** A program $t : f$ is **deterministic** if, in any state $S$, for any input configuration $I$, there exists at most one transition such that:

$$\forall S, I : \text{ if } \langle S \rangle t : f \xrightarrow{k'} \langle S' \rangle \text{ then } S' = S, f' = f, k' = k$$

This definition defines a unique transition by its resulting state, program fragment, and completion code. Observe that only the rewrite rules of the $\text{par}$ statement allow state $S$ to be changed in parallel by multiple transitions. The (par-1) rule aggregates the changes into a single state. The (par-4), (par-5), (par-6), and (par-7) rules use the semantic function $\text{Combine}$ (Algorithm 3) to combine the copies in the aggregated state. The (par-2) and (par-3) rules only allow one of the changed states to take effect. Before proving that all ForeC programs are deterministic, we prove that the aggregation of states and the semantic function $\text{Combine}$ are both deterministic. This is captured by Lemmas 1 and 2 below with the assumption that all the combine functions are deterministic.

**Definition 5.** A combine function $c$ is **deterministic** if, in any state $S$, for any three input values $\text{val}_1, \text{val}_2, \text{and} \text{val}_3$, there exists exactly one return value.
Hypothesis 1. The combine function $c$ always returns the same value regardless of the current state, provided that the input values, $\text{val}_1$, $\text{val}_2$, and $\text{val}_3$, are identical:

$$\forall S, S', I, t, \text{val}_1, \text{val}_2, \text{val}_3 :$$

$$\text{Eval}(S.E, I, t, c(\text{val}_1, \text{val}_2, \text{val}_3)) = \text{Eval}(S'.E, I, t, c(\text{val}_1, \text{val}_2, \text{val}_3))$$

Because the combine functions are defined in C, we require that the combine functions terminate without error.

Lemma 1. For any initial state $S = \langle E, A \rangle$, let $S' = \langle E', A' \rangle$ and $S'' = \langle E'', A'' \rangle$ be the states of two threads after their transition. If the threads can only change their own private variables and copies of shared variables, then the aggregation of $S'$ and $S''$ is deterministic if there exists only one aggregated state:

$$\forall S = \langle E, A \rangle, S' = \langle E', A' \rangle, S'' = \langle E'', A'' \rangle :$$

- if $E^A_1 = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'')$ and $E^A_2 = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'')$ then $E^A_1 = E^A_2$
- if $A^A_1 = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'')$ and $A^A_2 = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'')$ then $A^A_1 = A^A_2$

Proof. We begin by proving that the aggregation of environments $E'$ and $E''$ is deterministic. If the threads can only change their private variables and copies of shared variables, then their changes to $E$ are always mutually exclusive. That is, for any two threads $t'$ and $t''$, where $t' \neq t''$, the threads never access each other’s store because $E[t'] \neq E[t'']$. Moreover, by definition, the threads never access each other’s private variables in $E[G]$. Intersecting two environments, e.g., $E' \cap E$, always gives a new environment containing the variables that have the same values and statuses in $E'$ and $E$, i.e., have not changed. $E' \setminus (E' \cap E)$ always gives a new environment containing the variables that have changed in $E'$. The set operations are deterministic because two variables are either the same as each other or not. The aggregation always takes the union of the changes in $E'$ (i.e., $E' \setminus (E' \cap E)$) and in $E''$ (i.e., $E'' \setminus (E'' \cap E)$) with the unchanged variables in $E'$ and $E''$ (i.e., $E' \cap E''$). Because the changes in $E'$ and $E''$ are always mutually exclusive, the aggregation always takes the union of three disjoint environments.

We now prove that the aggregation of two sets of preemption statuses $A'$ and $A''$ is deterministic. Threads can only change $A$ by executing a status statement

---

$^7$ Recall from Chapter 3.3.2 that $E$ maps the global and thread scopes to their own store of variables, $E : \text{Id} \rightarrow \text{Store}$. Variables are mapped to a value and status, $\text{Store} : \text{Var} \leftrightarrow (\text{Val}, \text{Sts})$. The notation $E[t][\text{var}]$ looks up the value and status $(\text{val}, \text{sts})$ of thread $t$’s copy of var. Recall that $A$ maps the abort identifiers to their preemption statuses, $A : A \rightarrow \text{Val}$. The notation $A[a]$ looks up the preemption status $\text{val}$ of abort $a$. 

(the (status) rule). By construction, each status statement has a unique abort identifier \( a \). Thus, changes to \( A \) are always mutually exclusive. Intersecting two sets of preemption statuses, e.g., \( A' \cap A \), always gives a new set containing the statuses that have the same values in \( A' \) and \( A \), i.e., have not changed. \( A' \setminus (A' \cap A) \) always gives a new set containing the statuses that have changed in \( A' \). The set operations are deterministic because two statuses are either the same as each other or not. The aggregation always takes the union of the changes in \( A' \) (i.e., \( A' \setminus (A' \cap A) \)) and in \( A'' \) (i.e., \( A'' \setminus (A'' \cap A) \)) with the unchanged statuses in \( A' \) and \( A'' \) (i.e., \( A' \cap A'' \)). Because the changes in \( A' \) and \( A'' \) are always mutually exclusive, the aggregation always takes the union of three disjoint sets.

**Lemma 2.** If all combine functions are deterministic, then the semantic function \( \text{Combine} \) is deterministic if, in any state \( S = (E, A) \), for any three threads \( t_1, t_2, \) and \( t_0 \), there exists only one environment that can be returned:

\[
\forall S = (E, A), t_1, t_2, t_0 : \text{ if } E' = \text{Combine}(E, t_1, t_2, t_0) \text{ and } E'' = \text{Combine}(E, t_1, t_2, t_0) \text{ then } E' = E''
\]

**Proof.** The semantic function \( \text{Combine} \) is an algorithm that initializes all its local variables \( \text{preVal}, T, c, \) and \( \text{val} \), that is side-effect-free, and that uses only deterministic instructions. In particular, line 14 in Algorithm 3 is deterministic due to the hypothesis that all combine functions \( c \) are deterministic (Hypothesis 1). Hence, the semantic function \( \text{Combine} \) is deterministic.

**Theorem 2.** If all combine functions are deterministic, then all ForeC programs are deterministic.

**Proof.** The proof can be shown by a structural induction on \( t : f \).

**Base cases:** The (nop), (copy), (pause), and (status) rules imply that the following kernel statements have at most one transition:

\[
\begin{align*}
\langle S \rangle t : \text{nop} & \xrightarrow{0} \langle S \rangle t : \\
\langle S \rangle t : \text{copy} & \xrightarrow{0} \langle S' \rangle t : \text{nop} \\
\langle S \rangle t : \text{pause} & \xrightarrow{1} \langle S \rangle t : \text{copy} \\
\langle S \rangle t : \text{status}(a, \text{exp}) & \xrightarrow{0} \langle S' \rangle t : \text{nop}
\end{align*}
\]

The assignment, if-else, and while kernel constructs are each described by a pair of rewrite rules with complementary premises that do not depend on other transitions: (assign-shared) and (assign-private), (if-then) and (if-else), and (loop-then) and (loop-else). The premises are complementary in the sense that, if the premise of one rule is true, then the premise of the other rule must be false, and
vice versa. This implies that these kernel constructs have at most one transition:

\[
\begin{align*}
\text{if } \mathit{var} \in \mathit{GetShared}(t) \text{ then} & \quad (S) \ t: \mathit{var}=\mathit{exp} \xrightarrow{0} (S') \ t: \\
\text{otherwise} & \quad (S) \ t: \mathit{var}=\mathit{exp} \xrightarrow{0} (S'') \ t: \\
\text{if } \mathit{Eval}(S.E, I, t, \mathit{exp}) \neq 0 \text{ then} & \quad (S) \ t: \mathit{if} \ (\mathit{exp}) \ f_1 \ \mathit{else} \ f_2 \xrightarrow{1} (S) \ t: f_1 \\
\text{otherwise} & \quad (S) \ t: \mathit{if} \ (\mathit{exp}) \ f_1 \ \mathit{else} \ f_2 \xrightarrow{1} (S) \ t: f_2 \\
\text{if } \mathit{Eval}(S.E, I, t, \mathit{exp}) \neq 0 \text{ then} & \quad (S) \ t: \mathit{while} \ (\mathit{exp}) \ f \xrightarrow{1} (S) \ t: f; \mathit{while} \ (\mathit{exp}) \ f \\
\text{otherwise} & \quad (S) \ t: \mathit{while} \ (\mathit{exp}) \ f \xrightarrow{0} (S) \ t: \\
\end{align*}
\]

Of the rewrite rules considered in the base case, only the (copy), (status), (assign-shared), and (assign-private) rules make direct changes to state \( S \). The (copy) rule changes only the store \( E[t] \) of the executing thread \( t \). This can be verified by inspecting Algorithm 2 of the semantic function \( \text{COPY} \). By construction, each \text{status} statement has a unique \text{abort} identifier \( a \). Thus, the (status) rule never changes the status of the same \text{abort} identifier. The (assign-shared) rule changes only the store \( E[t] \) of the executing thread \( t \). The (assign-private) rule changes only the private variables in \( E[G] \) of the executing thread.

**Induction step:** The sequence operator (\( ; \)), \text{abort}, and \text{par} kernel statements are now of interest because they allow the composition of kernel constructs. For some \( t_1 : f_1 \) and \( t_2 : f_2 \) that are arbitrary compositions of kernel constructs, assume the induction hypotheses that they each have at most one transition:

\[
\begin{align*}
\text{If } & \exists S_1', S_1'', f_1', f_1'', k_1', k_1'' \text{ such that } (S_1) \ t_1 : f_1 \xrightarrow{k_1'} (S_1') \ t_1 : f_1' \quad (H3) \\
\text{and } & \exists S_1', S_1'', f_1', f_1'', k_1', k_1'' \text{ such that } (S_1) \ t_1 : f_1 \xrightarrow{k_1''} (S_1'') \ t_1 : f_1'' \ \\
\text{then } & S_1' = S_1'', \ f_1' = f_1'', \ k_1' = k_1'' \\
\end{align*}
\]

\[
\begin{align*}
\text{If } & \exists S_2', S_2'', f_2', f_2'', k_2', k_2'' \text{ such that } (S_2) \ t_2 : f_2 \xrightarrow{k_2'} (S_2') \ t_2 : f_2' \quad (H4) \\
\text{and } & \exists S_2', S_2'', f_2', f_2'', k_2', k_2'' \text{ such that } (S_2) \ t_2 : f_2 \xrightarrow{k_2''} (S_2'') \ t_2 : f_2'' \ \\
\text{then } & S_2' = S_2'', \ f_2' = f_2'', \ k_2' = k_2'' \\
\end{align*}
\]

Next, we show that the remaining sequence operator (\( ; \)), and the \text{abort} and \text{par} kernel statements have at most one transition.

1. Consider the fragment \( t_1 : f_1; f_2 \). Due to the induction hypothesis (H3),
there is only one possible transition for the program fragment $t_1 : f_1$, 

which is either $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1=0} \langle S_1' \rangle \ t_1 :$ 
or $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1 \in \{1,\ldots\}} \langle S_1' \rangle \ t_1 : f_1'$.

The table below shows that at most one sequence rule can be applied depending on the completion code $k_1$:

<table>
<thead>
<tr>
<th>$k_1$</th>
<th>0</th>
<th>1</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>(seq-right)</td>
<td>(seq-left)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, thanks to the induction hypothesis (H3), the sequence operator “;” is deterministic.

2. Consider the abort kernel statement in the fragment $t_1 : \text{weak? abort}(a_1, f_1)$. Due to the induction hypothesis (H3), there is only one possible transition for the program fragment $t_1 : f_1$, 

which is either $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1=0} \langle S_1' \rangle \ t_1 :$ 
or $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1 \in \{1,\ldots\}} \langle S_1' \rangle \ t_1 : f_1'$.

The table below shows that at most one abort rule can be applied depending on the completion code $k_1$ and the preemption status $A[a_1]$:

<table>
<thead>
<tr>
<th>$A[a_1]$ = 0</th>
<th>0</th>
<th>1</th>
<th>⊥</th>
<th>$A[a_1] \neq 0$</th>
<th>0</th>
<th>1</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>(abort-2)</td>
<td>(abort-1)</td>
<td>(abort-2)</td>
<td>(abort-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(abort-6)</td>
<td>(abort-3)</td>
<td>(abort-5)</td>
<td>(abort-4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, thanks to the induction hypothesis (H3), the abort kernel statement is deterministic.

3. Consider the par kernel statement in the fragment $t : \text{par}(t_1 : f_1, t_2 : f_2)$. Due to the induction hypotheses (H3) and (H4), there is only one possible transition for the program fragment $t_1 : f_1$, 

which is either $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1=0} \langle S_1' \rangle \ t_1 :$ 
or $\langle S_1 \rangle \ t_1 : f_1 \xrightarrow{k_1 \in \{1,\ldots\}} \langle S_1' \rangle \ t_1 : f_1'$.

and there is only one possible transition for the program fragment $t_2 : f_2$, 

which is either $\langle S_2 \rangle \ t_2 : f_2 \xrightarrow{k_2=0} \langle S_2' \rangle \ t_2 :$ 
or $\langle S_2 \rangle \ t_2 : f_2 \xrightarrow{k_2 \in \{1,\ldots\}} \langle S_2' \rangle \ t_2 : f_2'$.
The table below shows that at most one \texttt{par} rule can be applied depending on the completion codes $k_1$ and $k_2$:

<table>
<thead>
<tr>
<th>$k_1$</th>
<th>$k_2$</th>
<th>0</th>
<th>1</th>
<th>$\perp$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>(par-5)</td>
<td>(par-6)</td>
<td>(par-2)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>(par-7)</td>
<td>(par-4)</td>
<td></td>
</tr>
<tr>
<td>$\perp$</td>
<td></td>
<td>(par-3)</td>
<td>(par-1)</td>
<td></td>
</tr>
</tbody>
</table>

So, thanks to the induction hypotheses (H3) and (H4), the \texttt{par} kernel statement is deterministic.
3.6 Comparison with Esterel and Concurrent Revisions

This section compares ForeC with Esterel [45] and concurrent revisions [94] and Table 3.6 summarizes the qualitative comparison. ForeC is intended for applications that have control and data parallelism. Control parallelism is not a strength of concurrent revisions because its semantics does not consider (reactive) inputs and outputs. Unlike ForeC and Esterel, concurrent revisions allows an arbitrary number of asynchronous threads to be forked at runtime with the \texttt{rfork} construct. In ForeC and Esterel, the parent thread waits until all its child threads have terminated before it continues to execute. In concurrent revisions, the parent thread does not need to wait for its child threads to terminate, unless the \texttt{rjoin} construct is used. The semantics of concurrent revisions does not address how loops around the \texttt{rfork} or \texttt{rjoin} constructs affect the forking and joining of threads.

Similar to ForeC, threads in concurrent revisions communicate over shared variables. When a child thread is forked, it acquires a copy of the shared variables from its parent thread. When a child thread joins back with its parent, the copies from both threads are combined with a programmer-specified \textit{merge function}. The merge function always considers both copies, i.e., equivalent to ForeC’s combine policy \texttt{all}. Thus, thread communication is always delayed until the child thread terminates. In contrast, ForeC threads may execute over several ticks and thread communication is only delayed to the end of each tick. Esterel threads communicate instantaneously by emitting and receiving pure or valued signals during each tick.

<table>
<thead>
<tr>
<th>Property</th>
<th>Esterel</th>
<th>ForeC</th>
<th>Concurrent revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use for Parallelism</td>
<td>Control</td>
<td>Control and data</td>
<td>Data</td>
</tr>
<tr>
<td>Model of Computation</td>
<td>Synchronous</td>
<td>Asynchronous</td>
<td></td>
</tr>
<tr>
<td>Reactive Interface</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Parallelism is Dynamic</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Parallelism is Commutative and Associative</td>
<td>Yes</td>
<td>Depends on combine functions</td>
<td>Depends on merge functions</td>
</tr>
<tr>
<td>Thread Communication Method</td>
<td>Pure and valued signals</td>
<td>Shared Variables</td>
<td></td>
</tr>
<tr>
<td>Thread Communication Speed</td>
<td>Instantaneous</td>
<td>Delayed to the end of each tick</td>
<td>Delayed to thread termination</td>
</tr>
<tr>
<td>Resynchronization of shared variables or valued signals</td>
<td>Combine functions (\texttt{all values})</td>
<td>Combine functions with policies</td>
<td>Merge functions (\texttt{all values})</td>
</tr>
<tr>
<td>Preemption</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Causal Programs</td>
<td>Not always</td>
<td>Yes, by construction</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Comparison of ForeC with Esterel and concurrent revisions.
tick. Pure signals are either present or absent and carry no value. All potential signal emissions must be performed before the signal status (present or absent) can be read. This signal dependency requires the use of complex signal resolution mechanisms [46, 42, 48] by an implementation and limits the parallelism that can be achieved. Valued signals are like pure signals except each emission has an associated value. Before a valued signal can be read, all the emitted values are combined using a programmer-specified associative and commutative combine function. The combine function always considers all the copies, i.e., equivalent to ForeC’s combine policy all. Thus, the parallel construct for forking threads in Esterel is commutative and associative. For ForeC and concurrent revisions, the commutativity and associativity of their parallel construct depends on their combine and merge functions, respectively.

Preemptions in ForeC are inspired by Esterel, but behave slightly differently. Preemptions in Esterel are triggered instantaneously, whereas preemptions in ForeC are triggered with a delay of one tick. Concurrent revisions, however, does not support preemptions. Esterel programs may be non-causal [41] because of instantaneous feedback cycles. Thanks to delayed communication, ForeC and concurrent revisions programs are always causal by construction.

3.7 Discussion

This chapter has introduced the ForeC language that enables the deterministic parallel programming of multi-cores. The language features of ForeC help bridge the differences between synchronous-reactive programming and general-purpose parallel programming. Several well known parallel programming patterns were illustrated in ForeC to demonstrate its suitability for parallel programming. Importantly, ForeC makes deterministic parallelism accessible to traditional embedded C programmers. ForeC offers shared variable semantics that removes the burden of ensuring mutual exclusion from the programmer and ensures deadlock freedom. Thread isolation is guaranteed by stipulating that threads work on local copies of the shared variables. Resynchronizing the shared variables when the threads have finished their respective local ticks ensures program behavior that is agnostic to scheduling decisions. These features allow the local reasoning of each thread and simplifies the understanding and debugging of ForeC programs. Important definitions and proofs for ForeC were given for reactivity and determinism. Finally, a critical comparison showed that ForeC merges the benefits offered by synchronous languages, such as Esterel [45], with those offered by deterministic runtime solu-
3.7 Discussion

tions such, as concurrent revisions [94].

Traditional synchronous programming languages [41] are notoriously difficult to distribute or parallelize [46] due to their signal communication model. The key advantage of the model is that it removes the need to use thread synchronization mechanisms such as mutual exclusion. However, the need to maintain monotonic signal values [48, 49] makes it very difficult to parallelize these programs. Moreover, static analysis is needed to ensure that the presence or absence of all signals can be determined exactly in each tick of the program. In contrast, communication in ForeC is delayed using shared variables. The values of the shared variables are only resolved when threads complete their local ticks, hence allowing threads to execute in parallel and in isolation. Chapter 4 presents a straightforward compilation approach for ForeC and Chapter 5 presents a static timing analysis approach. Benchmarking in Chapter 6 reveals that our compilation approach offers good parallel execution that is amenable to static timing analysis. To our knowledge, no other synchronous language achieves parallel execution and timing predictability as good as ForeC.

ForeC’s combine functions are inspired by Esterel [118] but similar solutions can be found in other parallel programming frameworks, e.g., OpenMP’s reduction operators [36], MPI’s MPI_Reduce and MPI_Gather functions [37], Intel Thread Building Blocks’ tbb::parallel_reduce function and tbb::combinable data type [83], Intel Cilk Plus’ reducer data types [84], and Unified Parallel C’s collective functions [107]. Solutions developed for these frameworks could be reworked into ForeC combine functions. Appendix B provides more extensive examples of combine functions. A description of how the combine policies and combine functions work together to combine more than two copies of a shared variable is given.

All possible implementations of a ForeC program must adhere to ForeC’s formal semantics. This is unlike the deterministic runtime solutions developed for Pthreads [99, 100, 101, 102], OpenMP [103], and MPI [104], where determinism is only enforced at runtime and can be sensitive to changes in the program code. Moreover, program execution is not portable across the runtime solutions because each enforces its own notion of determinism.
The previous chapter described the new ForeC language for the deterministic and reactive parallel programming of CPSs. To be useful, the ForeC program must be compiled appropriately to exploit the parallelism of the target hardware architecture. This chapter describes how the ForeC compiler generates code for direct execution on the predictable parallel architectures described in Chapter 1.4. The chosen compilation strategy generates code that is amenable to static timing analysis and achieves good execution performance, as benchmarking results in Chapter 6 reveal.

4.1 Overview

The ForeC compiler can generate code for direct (bare metal) execution on the Xilinx MicroBlaze embedded multi-core (Chapter 1.4.1) and PTARM (Chapter 1.4.2) processors. Later in Chapter 4.9, we extend the compiler to generate code for execution on desktop multi-cores. Figure 4.1 is an overview of the compilation process.

Figure 4.1: Overview of compiling ForeC programs.
The first step is to check the syntax of the ForeC source code. This includes checking whether all threads have been defined and whether all variables accessed by multiple threads have been declared with the `shared` qualifier. The second step is to translate the ForeC statements into equivalent C code. Bootup and thread scheduling routines are generated for each core. The ForeC threads are statically allocated and statically scheduled on each core. The final step is to compile the generated C program with a GNU C compiler because GNU's computed `goto` extension is used to implement fast context-switching. This chapter describes the translation of ForeC to C code. For brevity, we omit input and output variables because we follow existing approaches [118] for creating the reactive interface.

### 4.2 Static Thread Scheduling

This section uses the example program shown in Figure 4.2a to illustrate the static scheduling generated by the ForeC compiler. The rest of this chapter deals exclusively with ForeC threads and, for brevity, we shorten “ForeC threads” into “threads”. Currently, the programmer statically allocates the threads to the cores and passes the allocations into the compiler. The scheduling is static and non-preemptive (cooperative). Thus, threads execute without interruption until they reach a context-switching point: a `par` or `pause` statement, or the end of their body.

The semantics of shared variables (see Chapter 3.1.3) ensures that threads execute their local ticks in isolation, e.g., independently of their siblings or their parent’s siblings. The compiler defines a total order (execution sequence) for all the threads, from which each core’s static thread scheduling order is derived. Currently, the total order is based on the depth-first traversal of the thread hierarchy. Figure 4.2b is the thread hierarchy of Figure 4.2a with numbers to indicate the total order. A lower number means higher execution priority. Figure 4.2c lists a possible thread allocation for two cores in their thread scheduling order. When a thread reaches a `par` statement, its child threads are forked for execution on their allocated cores.

The core that executes the parent thread is called the master core. The other cores that execute the child threads are called the slave cores. Depending on the thread allocations, a core could be the master core of one thread and be the slave core of another thread. For the `par` statement on line 6 of Figure 4.2a, core 1 is the master core and core 2 is the slave core. If thread `main` was allocated to core 2 instead, then core 2 would be the master core and core 1 would be the slave core.

Based on the thread allocation and scheduling order shown in Figure 4.2c, Figure 4.2d is a possible execution trace for Figure 4.2a. The trace for both cores
4.2 Static Thread Scheduling

shared int x=0 combine all with plus;
void main(void) {
    abort {
        x=x+1;
        pause;
        par(tA(),tB()); // id = 1
    } when (x > 1);
}
void tA(void) {
    par(tC(),tD()); // id = 2
}
void tB(void) {
    x=x+1;
    pause;
    x=x+1;
}
void tC(void) { int a=1; ... }
void tD(void) { ... }
int plus(int th1,int th2,int pre) {
    return (th1+th2);
}

(a) Example ForeC program.

(b) Total order.

Core 1     Core 2
main       tD
           tC

(c) Thread allocation.

(d) Possible execution trace of the compiled program.

Figure 4.2: Example ForeC program to be compiled.
### Table 4.1: Summary of the synchronization routines.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mFork</strong></td>
<td>Uses a non-blocking send to notify the slave cores whether or not the parent thread has forked.</td>
</tr>
<tr>
<td><strong>sFork</strong></td>
<td>Blocks until it receives whether or not the parent thread has forked.</td>
</tr>
<tr>
<td><strong>mJoin</strong></td>
<td>Blocks until it receives whether the child threads on other cores have terminated. Then, it notifies the slave cores whether the parent thread has resumed.</td>
</tr>
<tr>
<td><strong>sJoin</strong></td>
<td>Uses a non-blocking send to notify the master core whether or not its child threads have terminated. Then, it blocks until it receives whether the parent thread has resumed.</td>
</tr>
<tr>
<td><strong>mSync</strong></td>
<td>Synchronizes with all the cores, performs the housekeeping tasks, and then synchronizes with all the cores again to start the next global tick.</td>
</tr>
<tr>
<td><strong>sSync</strong></td>
<td>Synchronizes with all the cores and waits for the next synchronization to start the next global tick.</td>
</tr>
<tr>
<td><strong>mAbort</strong> and <strong>sAbort</strong></td>
<td>Evaluates the preemption condition of an abort.</td>
</tr>
</tbody>
</table>

(labeled “Core 1” and “Core 2”) progresses downwards from the top of Figure 4.2d. Thread executions are shown as white segments in the trace and each one has the thread’s name and the executed lines of code from Figure 4.2a. The compiler generates *synchronization routines* to manage the thread executions on the master and slave cores. These routines are shown as shaded segments in the trace and each one has the routine’s name. The names are prefixed with “m” or “s” to identify whether a routine is for a master or slave core, respectively. The names are suffixed with an integer to identify the routines that are used to manage the threads forked by the same parent thread. For example, the **mFork1**, **sFork1**, **mJoin1**, and **sJoin1** routines in Figure 4.2d all manage the threads forked by thread **main**. Table 4.1 summarizes the behavior of the routines. The **mFork** and **sFork** routines manage the forking of child threads (Chapter 4.4). The **mJoin** and **sJoin** routines manage the joining of child threads (Chapter 4.4). The **mSync** and **sSync** routines manage the global tick synchronization of all the cores (Chapter 4.8). In Figure 4.2d, the synchronization between the routines are shown as arrows marked with information that is sent. The information is an integer value that encodes the following execution states of a thread: 0 (TERM) for thread termination, 1 or greater for executing a *par* statement, and -1 (OTHER) for executing a *pause* statement or for not executing a *par* statement.

The threads and synchronization routines are statically scheduled on each core with (doubly) *linked lists*. Each node (defined in Figure 4.3a) of a linked list represents a thread or a synchronization routine and stores its continuation point (pc) and the links to its adjacent nodes (prev and next). A node’s pc is initially set to the start of the thread or routine’s body. Each core starts its scheduling
4.3 Structure of the Generated Program

This section describes the overall structure of the generated program and subsequent sections will describe each part in detail. Figure 4.4 shows a simplified version of the C program generated for the ForeC program in Figure 4.2a. All line numbers in the remainder of this chapter refer explicitly to Figure 4.4. The generated C program contains:

- The global declarations and functions from the ForeC program (lines 4–8).
- The global declarations for storing the execution states of the threads and implementing the shared variables (lines 11–15).
- The main function (line 18) with the bootup routine (lines 36–43), synchronization routines (lines 46–141), and threads (lines 144–190).

When the cores enter the main function, they execute the bootup routine to initialize their linked lists. First, a node is created for each thread and each synchronization routine (lines 20–34). Second, the nodes are linked together to create the by jumping to the pc of its first node. When a context-switching point is reached during the execution of a thread or routine, a jump is made to the pc of the next node. A core will only execute the threads and routines in its linked list. Thus, inserting or removing a thread or routine from the list controls whether it is included or excluded, respectively, from execution. Figures 4.3b and 4.3c illustrate the insertion and removal operations defined in Figure 4.3a. The remainder of this chapter describes how a ForeC program is compiled into a C program and how the linked lists are created and used to implement the ForeC semantics.

![Figure 4.3: Definition of a linked list node and its operations.](image-url)
When the program starts

<table>
<thead>
<tr>
<th>Execution Point</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>When the program starts</td>
<td>Core 1: [main] [mFork1] [mSync]</td>
</tr>
<tr>
<td></td>
<td>Core 2: [sFork1] [sSync]</td>
</tr>
</tbody>
</table>

When main forks (id = 1)

<table>
<thead>
<tr>
<th>Execution Point</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>When main forks (id = 1)</td>
<td>Core 1: [mAbort1] [tA] [mFork2] [mJoin1] [mSync]</td>
</tr>
<tr>
<td></td>
<td>Core 2: [sAbort1] [sFork2] [sB] [sJoin1] [sSync]</td>
</tr>
</tbody>
</table>

When tA forks (id = 2)

<table>
<thead>
<tr>
<th>Execution Point</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>When tA forks (id = 2)</td>
<td>Core 1: [mAbort1] [tC] [mJoin2] [mJoin1] [mSync]</td>
</tr>
<tr>
<td></td>
<td>Core 2: [sAbort1] [sD] [sJoin2] [sB] [sJoin1] [sSync]</td>
</tr>
</tbody>
</table>

Table 4.2: Core 1 and 2’s initial lists and subsequent lists when threads fork.

initial linked list for each core (lines 36–43). These initial lists are visualized by the second row of Table 4.2. The threads and routines are inlined into the main function because fast context-switching is implemented by jumping between C labels with GNU’s computed goto extension. Jumping with goto is restricted to C labels located in the same function scope. To avoid the need to create stacks for each thread to maintain their local variables, the local variables are given unique names and hoisted up to the global scope (e.g., tC’s local variable a on line 5). However, functions executed on the same core will share the same stack space. To avoid stack corruption, all the functions must execute atomically, i.e., without interruption. In future versions of the compiler, we wish to remove this limitation by creating proper thread stacks.

### 4.4 The par Statement

The execution of a ForeC program starts with its main thread. The slave cores must wait for their allocated threads to be forked. The timing of when threads fork and join can only determined at runtime. Hence, before a core executes a thread, it must check that no other higher priority thread allocated to it will be forked. Otherwise, the higher priority thread must be executed first. This is achieved by scheduling an mFork routine after a parent thread completes its local tick. It uses a non-blocking send to notify the slave cores whether or not the parent thread has forked. Thus, a slave core uses an sFork routine to block until it receives whether or not the parent thread has forked. To ensure correct scheduling order, the sFork routine has the same execution priority as the parent thread. When a fork does occur, the mFork and sFork routines instruct their cores to suspend the parent thread and to schedule the child threads. In the first global tick of Figure 4.2d, mFork1 notifies sFork1 that thread main has not forked (OTHER is sent). In the
```c
#include "node.h" // Figure 4.3a

// Programmer–defined
int x=0; // Shared variable
int *tC; // tC's local variable
int plus (int th1, int th2, int pre) {
    return (th1+th2);
}

// Compiler–defined
enum State {OTHER=-1,TERM=0};
int mainState=OTHER, tAState=OTHER, tBState=OTHER, tDState=OTHER;
int x_main, x_tA, x_tB, x_tC, x_tD;

// Entry point
void main(void) {
    // Nodes for the linked lists
    Node mSync=
    Node mAbort1=
    Node mJoin1=
    Node mJoin2=
    Node mFork1=
    Node mFork2=
    Node sSync=
    Node sAbort1=
    Node sJoin1=
    Node sJoin2=
    Node sFork1=
    Node sFork2=

    // Create initial linked lists
    mainState=OTHER; send(mainState);
    if (mainState == OTHER) {
        mFork1: {
            send(tAState);
            if (tAState == OTHER) {
                insert (tA, mFork2); insert (mFork2, tC);
                remove(mFork1); remove(tA); remove(mFork2); goto +tC.pc;
            } else { goto +mFork2.next->pc; }
        }
        mFork2: {
            send(tAState);
            if (tAState == OTHER) {
                insert (tA, mFork2); insert (mFork2, tC);
                remove(tA); remove(mFork2); goto +tC.pc;
            } else { goto +mFork2.next->pc; }
        }
        mJoin2: {
            receive (tDState);
            if (tDState == OTHER) {
                insert (tD, sJoin2); remove(sJoin2);
                goto +tD.pc;
            } else { goto +sFork2.next->pc; }
        }
    } else { goto +tA.pc; }
    else { goto +sJoin2.next->pc; }
    }
    sJoin2: {
        send(tDState); receive (tAState);
        if (tAState == OTHER) { remove(sJoin2); }
        goto +sJoin2.next->pc;
    }
    }
    sJoin1: {
        receive (tBState);
        send(tBState); receive (mainState);
        if (mainState == OTHER) { remove(sJoin1); }
        goto +sJoin1.next->pc;
    }
    }
}

// Preempting
mAbort1: {
    if (x > 1) {
        Remove the following nodes: tA, mFork2,
        mJoin1, tC, mJoin2, and mAbort1.
    }
}
```

Figure 4.4: Example of the C program generated for Figure 4.2a.
main.pc = &&abort1; goto main.pc;
} else { goto +main.next->pc; }
sAbort1: 
if (x > 1) {
Remove the following nodes: sFork2,tB, sJoin1,tD,sJoin2, and sAbort1.
goto +sAbort1.next->pc;
} else { goto +sAbort1.next->pc; }
sSync: { 
    barrier (); emitOutputs(); sampleInputs();
    goto +sSync.next->pc;
}
// Threads
main: {
    copy(x.main,x);
    /\ abort \\ /
    x.main=1;
}
// pause;
main.pc= &&pause1;
goto +main.next->pc;
pause1:;
if (x > 1) { goto abort1; }
copy(x.main,x);
}

// par(tA,tB) with id=1
mainState=1; main.pc= &&&join1;
goto +main.next->pc;
join1:;
copy(x.main,x);
}
// when (x.main > 1);
abort1: exit (0);
}
// par(tC,tD) with id=2
tAState=2; tA.pc= &&&join2; goto mFork2;
join2:;
// Termination.
tAState=TERM;
remove(tA);
goto +tA.next->pc;
}
tB: {
    copy(x.tB,x.main);
    x.tB=x.tB+1;
    /
    x.tB=x.tB+1;
    // Termination.
    xB=x.tB;
    /
    x.tB=x.tB+1;
    // Termination.
tBState=TERM; remove(tB);
goto +tB.next->pc;
}
tC: { a.tC=1; ... }
tD: { ... }
}
// End of main()
4.4 The par Statement

second global tick, \texttt{mFork1} notifies \texttt{sFork1} that thread \texttt{main} has forked (1 is sent).

Before a core executes a parent thread that was suspended by a fork, it must check that all of its child threads have terminated. This is achieved by scheduling an \texttt{mJoin} routine after the child threads have completed their respective local ticks. It \textit{blocks} until it receives whether or not the child threads on the slave cores have terminated. When all child threads have terminated, the \texttt{mJoin} routine instructs the master core to resume the parent thread. Thus, each slave core schedules an \texttt{sJoin} routine after its child threads complete their respective local ticks. It uses a \textit{non-blocking send} to notify the master core whether or not the child threads on the slave core have terminated. In the second global tick of Figure 4.2d, \texttt{sJoin1} notifies \texttt{mJoin1} that thread \texttt{tB} has not terminated (OTHER is sent). In the third global tick, \texttt{sJoin1} notifies \texttt{mJoin1} that thread \texttt{tB} has terminated (TERM is sent).

We now describe the C code that is generated for each \texttt{par} statement and how the synchronization routines are incorporated into the linked lists. Each \texttt{par} statement is assigned a unique positive integer \texttt{id} by the compiler. Lines 157–160 in Figure 4.4 is an example of the C code that is generated for a \texttt{par} statement. Line 158 sets the parent thread’s execution state to \texttt{id} and sets the parent thread’s \texttt{pc} to be immediately after the \texttt{par} statement. Line 159 is a context-switch to the parent thread’s \texttt{mFork} routine. Lines 46–54 is an example of the C code that is generated for an \texttt{mFork} routine. Line 47 sends the parent thread’s execute state to the slave cores. If the parent thread has forked, then lines 49–52 insert the allocated child threads and an \texttt{mJoin} routine into the linked list. The parent thread and \texttt{mFork} routine are removed from the linked list. If a child thread can fork its own threads, then further \texttt{mFork} and \texttt{sFork} routines need to be inserted into the linked lists. This ensures that the nested threads can be forked. Line 52 is a context-switch to the first node that was inserted. Otherwise, if the parent thread has not forked, then line 53 is a context-switch to the next node in sequence. Recall that the slave cores have an \texttt{sFork} routine in their initial linked list. Lines 55–63 is an example of the C code that is generated for an \texttt{sFork} routine. Line 56 blocks until it receives whether the parent thread has forked. If the parent thread has forked, then lines 58–60 insert the allocated child threads and an \texttt{sJoin} routine into the linked list. The \texttt{sFork} routine is removed from the linked list. Line 61 is a context-switch to the first node that was inserted. Otherwise, if the parent thread has not forked, then line 62 is a context-switch to the next node in sequence. The last two rows of Table 4.2 visualizes core 1 and 2’s linked lists when threads \texttt{main} and \texttt{tA} fork (lines 157 and 166 respectively).
Lines 169–172 in Figure 4.4 is an example of the C code that is generated for the end of a child thread to handle thread termination. Line 170 sets the thread’s execution state to \texttt{TERM}. Line 171 removes the thread from the linked list. Line 172 is a context-switch to the next node in sequence. Lines 97–108 is an example of the C code that is generated for an \texttt{mJoin} routine. Line 98 blocks until it receives the execution state of each child thread. If all the child threads have terminated, then line 102 sets the execution state of the parent thread to \texttt{OTHER} and sends that state to the slave cores. Lines 103–104 insert the parent thread back into the linked list and removes the nodes associated with the \texttt{par} statement. This is followed by a context-switch to the parent thread. Otherwise, if some child threads have not terminated, then line 106 is a context-switch to the next node in sequence. Lines 109–114 is an example of the C code that is generated for an \texttt{sJoin} routine. Line 110 sends the execution state of each child thread to the master core. Line 111 blocks until it receives whether the parent thread has been resumed. If the parent thread has been resumed, then line 112 removes the \texttt{sJoin} routine from the linked list. Line 113 is a context-switch to the next node in sequence.

4.5 The \texttt{pause} Statement

The \texttt{pause} statement is a context-switching point and lines 150–153 in Figure 4.4 is an example of the C code that is generated. Line 151 sets the current thread’s \texttt{pc} to be immediately after the \texttt{pause} statement. Line 152 is a context-switch to the next node in sequence. In the next global tick, execution will resume from statement immediately after the \texttt{pause} statement.

4.6 Shared Variables

Shared variables are hoisted up to the program’s global scope to allow all cores to access them (e.g., line 4 in Figure 4.4). The copies of shared variables are implemented as unique global variables (e.g., line 15) to allow them to be combined on different cores. In each thread, all shared variable accesses are replaced by accesses to their copies (e.g., lines 148 and 177). The shared variables are copied at the start of each local tick, i.e., start of each thread body, and after each \texttt{pause} and \texttt{par} statement. For example, the shared variable \texttt{x} on line 4 is copied by thread \texttt{main} on lines 145, 155 and 161. As defined by the (par-4), (par-5), (par-6), and (par-7) semantic rules given in Chapter 3.3.4, the \texttt{par} statement is responsible for combining the copies of shared variables. More precisely, when the child threads
of a `par` statement complete their respective local ticks, their copies of shared variables are combined. The combined result is assigned to their parent thread. This combine process is implemented by the `mJoin` routine (e.g., line 99) because it waits for the child threads to complete their respective local ticks. The final values of the shared variables are computed by the `mJoin` routine of thread `main`.

4.7 The `abort` Statement

We begin by describing the C code that is generated for an `abort` that does not have the optional `immediate` or `weak` keywords. Conditional jumps, using the preemption condition, are inserted after each `pause` statement in the `abort` body. For example, lines 147–162 in Figure 4.4 is an `abort` and a conditional jump is inserted on line 154 after the `pause` statement. The preemption condition `x>1` is used in the conditional jump. If the preemption condition evaluates to `true`, a jump is made to the statement immediately after the `abort` (e.g., line 163). If a `par` statement is inside the `abort` body, then the preemption condition must be evaluated before the threads can execute. For example, in the third global tick of Figure 4.2d, the cores use the `mAbort` and `sAbort` routines to evaluate the preemption condition on line 7 of Figure 4.2a. It is safe to evaluate the preemption conditions in parallel because they are side-effect free by definition (Chapter 3.1.6). Thus, when a fork occurs, an `Abort` routine is inserted before the child threads in the linked lists. For a master core, lines 117–123 in Figure 4.4 is an example of the C code that is generated for an `mAbort` routine. Line 118 evaluates the preemption condition. If it evaluates to `true`, then line 119 removes the nodes associated with the `par` statement. Line 121 sets the parent thread’s `pc` to be immediately after the `abort` statement and context-switches to the parent thread. Otherwise, if the preemption condition evaluates to `false`, then line 122 is a context-switch to the next node in sequence. For a slave core, lines 124–130 is an example of the C code that is generated for an `sAbort` routine and is similar to that of an `mAbort`. Line 125 evaluates the preemption condition. If it evaluates to `true`, then line 126 removes the nodes associated with the `par` statement. Line 128 is a context-switch to the next node in sequence. Otherwise, if the preemption condition evaluates to `false`, then line 129 is a context-switch to the next node in sequence.

The optional `immediate` keyword allows the preemption condition to be evaluated before the `abort` body is executed for the first time. Thus, an additional conditional jump, using the preemption condition, is inserted at the start of the `abort` body. Figure 4.5a is an example of the C code that would be generated if
Compiling ForeC for Parallel Execution

100

/* abort */
if (x > 1) { goto abort1; }
x_main=1;

// pause;
main.pc=&&pause1;
goto *main.next->pc;
pause1;;
if (x > 1) { goto abort1; }
copy(x_main,x);

// par(tA,tB) with id=1
mainState=1;
goto *main.next->pc;
join1 ;;
copy(x_main,x);
} // when (x_main > 1);

int triggered = 0;
/* abort */
if (triggered) { goto abort1; }
x_main=1;

// pause;
main.pc=&&pause1;
goto *main.next->pc;
pause1;;
triggered = (x > 1);
copy(x_main,x);

// par(tA,tB) with id=1
mainState=1;
goto *main.next->pc;
join1 ;;
copy(x_main,x);
} // when (x_main > 1);

(a) Immediate and strong abort.

int triggered = 0;
/* abort */
if (x > 1) { goto abort1; }
x_main=1;

// pause;
main.pc=&&pause1;
goto *main.next->pc;
pause1;;
triggered = (x > 1);
copy(x_main,x);

// par(tA,tB) with id=1
mainState=1;
goto *main.next->pc;
join1 ;;
copy(x_main,x);
} // when (x_main > 1);

(b) Non-immediate and weak abort.

int triggered = 0;
/* abort */
if (triggered) { goto abort1; }
x_main=1;

// pause;
main.pc=&&pause1;
goto *main.next->pc;
pause1;;
triggered = (x > 1);
copy(x_main,x);

// par(tA,tB) with id=1
mainState=1;
goto *main.next->pc;
join1 ;;
copy(x_main,x);
} // when (x_main > 1);

(c) Immediate and weak abort.

Figure 4.5: C code for the immediate and weak variants of the abort on lines 147–162 of Figure 4.4.

the abort on lines 147–162 in Figure 4.4 was an immediate abort. The optional weak keyword delays the jumping to the end of the abort body when the preemption condition evaluates to true. Thus, the conditional jump is separated into two parts: 1) the evaluation of the preemption condition and 2) the resulting jump. The evaluation is inserted directly after each pause statement and the jump is inserted directly before each pause statement. If a par statement is inside the weak abort, then the mAbort and sAbort routines are inserted after the child threads in the linked lists. Figure 4.5b is an example of the C code that would be generated if the abort on lines 147–162 in Figure 4.4 was a weak abort. Figure 4.5c is an example of the C code generated if it was an immediate and weak abort.

4.8 Global Tick Synchronization

The notion of a global tick is preserved by ending each linked list with a Sync routine that implements barrier synchronization. This synchronization is shown at the end of each global tick in Figure 4.2d. For the master core that executes the main thread, lines 133–138 is an example of the C code that is generated for an mSync routine. Line 134 is a barrier synchronization for the end of the tick. Line 135 performs the following housekeeping tasks: finalizing the values of the shared variables, emitting outputs, and sampling inputs. Line 136 is a barrier synchronization to signal the start of the next global tick. Line 137 is a context-switch to the first node in the linked list. For the remaining slave cores,
4.9 Generating Programs for Desktop Execution

This section describes how the ForeC compiler is extended to generate executable code for desktop multi-cores. Desktop systems use an operating system to manage the execution of programs on the cores. To utilize multiple cores, a program must create multiple threads that the operating system can schedule. We modify the ForeC compiler to generate a Pthread for each core in the system. Each Pthread is responsible for executing the ForeC threads statically allocated to the same core, as shown in Figure 4.6. In effect, a fixed pool of Pthreads executes the ForeC threads and the cost of creating each Pthread is only incurred once. Although the Pthreads will be dynamically scheduled by the operating system, the original ForeC threads will still follow their static schedule. For future work, a better scheduling method for desktops can be developed. Finally, the generated Pthreads program is compiled with a GNU C compiler.

For the ForeC program of Figure 4.2a, Figure 4.7 is a simplified extract of the generated Pthreads program. In addition to the global declarations shown in Figure 4.4, there are now Pthreads-related declarations (lines 2–3) and a new main function for creating the Pthreads (line 6). The original main function from Figure 4.4 (line 18) is renamed as forecMain (line 11). When the operating system executes the main function, the Pthreads start executing the forecMain function and, hence, the statically allocated ForeC threads.

Figure 4.6: Using Pthreads to adapt the generated code for desktop multi-cores.

lines 139–141 is an example of the C code that is generated for an sSync routine. Line 140 are barrier synchronizations for the end of the tick and the start of the next tick. This is followed by a context-switch to the first node in the linked list.

4.9 Generating Programs for Desktop Execution

This section describes how the ForeC compiler is extended to generate executable code for desktop multi-cores. Desktop systems use an operating system to manage the execution of programs on the cores. To utilize multiple cores, a program must create multiple threads that the operating system can schedule. We modify the ForeC compiler to generate a Pthread for each core in the system. Each Pthread is responsible for executing the ForeC threads statically allocated to the same core, as shown in Figure 4.6. In effect, a fixed pool of Pthreads executes the ForeC threads and the cost of creating each Pthread is only incurred once. Although the Pthreads will be dynamically scheduled by the operating system, the original ForeC threads will still follow their static schedule. For future work, a better scheduling method for desktops can be developed. Finally, the generated Pthreads program is compiled with a GNU C compiler.

For the ForeC program of Figure 4.2a, Figure 4.7 is a simplified extract of the generated Pthreads program. In addition to the global declarations shown in Figure 4.4, there are now Pthreads-related declarations (lines 2–3) and a new main function for creating the Pthreads (line 6). The original main function from Figure 4.4 (line 18) is renamed as forecMain (line 11). When the operating system executes the main function, the Pthreads start executing the forecMain function and, hence, the statically allocated ForeC threads.
4.10 Discussion

This chapter has presented the compilation of ForeC programs for direct execution on parallel hardware architectures. The ForeC compiler is a command-line tool implemented using Flex and Bison with 79 C++ supporting classes. The ANSI C syntax and grammar are extended with the ForeC keywords. The front-end checks the ForeC and C syntax and generates an intermediate format that captures the parallelism, preemption scopes, and local tick boundaries. Errors are generated when threads, shared variables, or combine functions are undefined, or when the wrong types are detected in the arguments of the `par` and `abort` statements. The provided thread-to-core allocation is also checked. The middle-end checks all variables for the number of threads that may access them. Errors are generated when private variables need to be declared as `shared` variables. Each thread is marked in the intermediate format with their allocated core to help determine the required synchronization routines. The back-end generates C code for the threads, synchronization routines, and bootup routine for the cores. An XML file is also generated to contain program information for static analysis, and a Graphviz diagram of the program’s control-flow graph. The following key features of ForeC are still being implemented: `shared` arrays, C operator short-hands, `all` and `new` combine policies, and weak and non-immediate aborts. The code generation is structural, meaning that a nesting of ForeC constructs is compiled into a nesting of each construct’s generated code. Although this structured approach to compilation lends well to static analysis, it can limit the opportunities for optimizing the code for binary size, memory usage, and execution time. For example, it is possible that compiler optimizations, such as loop unrolling, dead code elimination, and code motion can lead to simplified programs that are easier to analyze and have shorter WCRTs. The use of templates for generating C code from ForeC constructs can lead to code bloat because the templates must be generic enough to cover all
4.10 Discussion

possible variants of a ForeC construct. Light-weight synchronization routines are generated to manage the forking and joining of threads across the cores. The use of linked lists to manage the scheduling of threads and routines is inspired by that of the Columbia Esterel Compiler [42]. The advantages with our static scheduling approach include: 1) light-weight scheduling of ForeC threads and 2) easier to analyze because all scheduling decisions are known beforehand. However, the disadvantages include: 1) inability to dynamically load balance the ForeC threads to utilize the idle cores and 2) needing to recompile the program to target a different number of cores.

The distribution of traditional synchronous programs over multiple processors is not new [46, 131, 132, 47, 48, 133, 134]. It is motivated by the desire to execute computations closer to their inputs and outputs, which may be distributed over a geographical area. Unfortunately, the use of signals for instantaneous communication makes compilation notoriously difficult. First, causality analysis [118] is needed to ensure that the presence or absence of all signals can be determined exactly in each global tick. Second, the compiler must generate code for resolving signal statuses at runtime. The common approach is to compile away the threads and generate a sequential program [42, 118]. Third, the sequential program is partitioned into subprograms and distributed to execute on their allocated processors. Desynchronization techniques [136, 137, 138] can be used when the processors execute and communicate at different speeds. SynDEx [135] is a tool that automatically distributes synchronous programs and considers the cost of communication between the processors. In contrast, ForeC is significantly easier to compile because thread communication is delayed with shared variables (Chapter 3.1.3). Causality analysis is not required and ForeC threads can be distributed directly to the available cores. The parallelism specified by the programmer is preserved by the ForeC compiler and a sequential intermediate format is not required.

With the advent of multi-cores, the distribution of synchronous programs is motivated by the desire to improve their execution performance. The distribution of synchronous programs over multi-threaded and multi-core reactive processors has been studied [196, 197, 198, 199, 133]. Reactive processors handle the scheduling of threads in hardware, thereby simplifying the code generation. However, causality analysis is still required and signal statuses still need to be resolved at runtime. Signal resolution may reduce a program’s parallel performance because a thread must wait for a signal’s status to be resolved before it can be read. There have been studies on the parallelization of synchronous programs on general-purpose multi-cores [47, 48, 49]. These approaches extract a parallel program from a se-
quential representation of the original synchronous program. Due to control and signal dependencies, the opportunities for extracting parallelism from a sequential program is limited. In contrast, execution dependencies only exist at the local tick boundaries of ForeC threads. Hence, ForeC threads have more opportunity to execute in parallel.
Static WCRT analysis is needed to verify that an implementation of a synchronous program satisfies the synchrony hypothesis. The previous chapter described the compilation of ForeC programs for direct execution on parallel hardware architectures. The main advantage of this compilation strategy is that an operating system is not needed to manage the execution of the threads. This simplifies WCRT analysis because the complexities and nuances of an operating system do not need to be considered. This chapter presents the ForeCast static timing analyzer developed to compute tight WCRTs of ForeC programs deployed on multi-cores. Using an intermediate format to represent the program’s control-flow, ForeCast uses the reachability technique to explore all the possible execution paths. During the exploration, the execution time of every tick is computed and the longest is the computed WCRT. To compute the WCRT precisely, ForeCast considers the effects of inter-core interference, thread-level parallelism, software-based synchronization, and thread scheduling.

5.1 Overview

Figure 5.1 shows the main steps that the ForeCast analyzer takes to compute the WCRT of a ForeC program. We assume that the C-code generated by the ForeC
The ForeCast Static Timing Analyzer
compiler is compiled into an executable binary with optimizations that affect the
control-flow disabled. This is a common approach [9, 23] for preserving the control-
flow of the C-code in the binary and allows the computed execution times to be
mapped accurately back to the C-code. It is beyond the scope of this thesis to
study the effects of compiler optimizations on the generated binary. The first step
is to disassemble the binary of the ForeC program into human readable assembly
code. The second step is to reconstruct the concurrent control-flow graph (CCFG)
from the assembly code. The CCFG is inspired by PRET-C’s intermediate for-
mat [200] and captures the following aspects of the control-flow: data computation,
conditional branching, parallelism, preemption, and barrier synchronization. To
facilitate the reconstruction of the CCFG, the ForeC compiler annotates the C-
code it generates with contextual information, e.g., whether the C-code is for a
par, abort, or pause statement, or a synchronization routine. The annotations
are implemented as C-comments and are preserved by compiling with debugging
enabled. The third step is to use the reachability technique to find all the pos-
sible ticks in the CCFG. Using a timing model of the hardware architecture, the
execution time of each tick is computed as the CCFG is traversed. The following
challenges must be tackled simultaneously when computing the execution time on
multi-cores: 1) an access to a shared hardware resource could be delayed by a
simultaneous access from another core, 2) the execution times of threads on sepa-
rate cores could overlap, 3) software-based synchronization could cause the cores
to block for an arbitrary amount of time, and 4) the thread scheduling determines
when threads are executed. Our proposed ForeCast analyzer is the first to consider
all these factors for synchronous programs. The longest execution time computed
for any tick is the program’s computed WCRT.

The literature review (Chapter 2.3) showed that a number of works have
modeled different aspects of a multi-core system in some detail: pipelines [227],
caches [224, 226], and shared buses [225, 227, 228]. However, not many have
focused on analyzing the effects that software-based thread synchronization and
scheduling have on the computed WCET or WCRT. These features are essen-
tial for the realistic implementation of parallel programs. This thesis focuses on
addressing these gaps to offer a static timing analysis solution that can analyze re-
alistic parallel programs. Without loss of generality, we assume that the programs
are deployed on time predictable parallel architectures and have straightforward
timing models, like those described in Chapter 1.4. Based on existing time pre-
dictable parallel architectures [31, 161, 189, 190], we assume that each core has
private data and instruction scratchpads. For each core, the scratchpads are stat-
ically allocated with the instructions it needs to execute and the private variables of its allocated threads. All the cores connect via a shared TDMA bus to global memory that contains only shared variables. More complex architectures can be analyzed as long as a suitable timing model is available. The following sections describe the CCFG, shared TDMA bus, and reachability technique in more detail.

5.2 Concurrent Control-Flow Graph

Figure 5.2 is a simplified example of how the CCFG is reconstructed for a ForeC program. Figure 5.2a shows the main function of a ForeC program and Figure 5.2b is the generated C-code with C-comments to identify the code of each par, abort, or pause statement. This provides the necessary high-level information for resolving the destinations of the context-switches and blocking times of the synchronizations. The annotated C-code is compiled with debugging enabled and optimizations disabled to preserve the C-comments and the control-flow. Figure 5.2c shows the result of disassembling the executable binary into assembly code and Figure 5.2d shows the reconstructed CCFG. In addition to the usual data computation and conditional branching nodes of a CFG, the CCFG uses extra nodes to capture parallelism (pair of triangles with the threads’ control-flow in between), preemption (pair of marked diamonds and directed edge to demarcate the abort body), and barrier synchronization (black node). In Figure 5.2d, each node is annotated with line numbers that correspond to the assembly instructions. Note that the synchronization routines are excluded from the CCFG. This is because the program’s runtime state determines what routines need to be executed at each context switching point on each core. Showing such dynamic information in the CCFG is impractical. Instead, the execution times of the routines are modeled internally by the ForeCast analyzer and Chapter 5.4 explains this in detail.

5.3 Shared TDMA Bus

During the execution of a program’s tick, a core may need to fetch data from its private data scratchpad or from the global memory. Accesses to its private data scratchpad complete in a constant number of clock cycles because there are no contentions with other cores. However, accesses to global memory must go through the shared TDMA bus and, therefore, may experience a variable time delay. For each core $n$, the TDMA bus allocates a time slot that is $s_n$ clock cycles long for the core’s dedicated use of the bus. Let $S = \{s_1, \ldots, s_N\}$ be the set of
Figure 5.2: Constructing the CCFG of a ForeC program.
time slots, where $N$ is the total number of cores. The slots are statically scheduled from $s_1$ to $s_N$ to create a static bus schedule. Figure 5.3a is an example of a bus schedule for two cores, where core 1’s time slot $s_1$ and core 2’s time slot $s_2$ are both 4 clock cycles long. The following equations compute the start time $s_{\text{start}}^n$ and end time $s_{\text{end}}^n$ of core $n$’s time slot, relative to the start of the bus schedule:

$$s_{\text{start}}^n = s_{\text{start}}^{n-1} + s_{n-1}$$ \hspace{1cm} \text{where } s_{\text{start}}^1 = 1 \hspace{1cm} \text{(5.1)}$$

$$s_{\text{end}}^n = s_{\text{start}}^{n} + s_{n} - 1 \hspace{1cm} \text{(5.2)}$$

The length of the bus schedule is equal to the end time of the last core’s slot, i.e., $s_{\text{end}}^N$. The bus schedule is repeated indefinitely at runtime. Each time a core needs to access global memory, it requests the use of the bus. Figure 5.3b exemplifies how the bus schedule of Figure 5.3a affects core 1’s memory accesses time. Core 1’s execution time is shown below the bus schedule and up arrows indicate when memory accesses are encountered and down arrows indicate when they are completed. The total access time is indicated by a corresponding shaded bar. For simplicity, we assume that a global memory access requires two clock cycles. If the request occurs during the core’s slot, then it is granted immediately (first memory access in Figure 5.3b). Otherwise, the request is granted the next time the core’s slot is scheduled (second memory access in Figure 5.3b). If the core cannot complete its global memory access before its slot expires, then the access is aborted and must be retried at the core’s next slot (third memory access in Figure 5.3b). This means that the TDMA slots cannot be shorter than the memory access time, otherwise the access will never complete. Thus, a core experiences a bus delay when its request falls outside of its scheduled slot or its memory

Figure 5.3: Timing behavior of a shared TDMA bus.
access fails to complete before its slot expires. To compute the bus delay, we need to determine when the bus request occurs in relation to the core’s slot in bus schedule. Because the bus schedule is repeated indefinitely, the bus request time can be divided by the length of the bus schedule and the remainder is the bus request time since the last repetition of the bus schedule. For example, the third bus request in Figure 5.3b occurs 12 \((\text{mod } 8) = 4\) clock cycles into the second repetition of the bus schedule. Let \(t_m\) be the memory access time and \(t_n\) be core \(n\)’s bus request time. Then, \(t_r = t_n \text{ (mod } s_{N}^{\text{end}}\) is core \(n\)’s bus request time since the last repetition of the bus schedule, where \(s_{N}^{\text{end}}\) is the length of the bus schedule. Then, the bus delay for core \(n\) is calculated as:

\[
\text{Delay}(n, t_r, t_m) = \begin{cases} 
  s_{n}^{\text{start}} - t_r & \text{If } t_r < s_{n}^{\text{start}} \\
  0 & \text{If } s_{n}^{\text{start}} \leq t_r \text{ and } (t_r + t_m) \leq s_{n}^{\text{end}} \\
  s_{n}^{\text{end}} - t_r + s_{n}^{\text{start}} & \text{If } s_{n}^{\text{end}} < (t_r + t_m)
\end{cases}
\] (5.3)

The delay is \(s_{n}^{\text{start}} - t_r\) if the core’s bus request occurs before its time slot (second memory access in Figure 5.3b). The delay is 0 if the core’s bus request occurs during its time slot and the bus access can be completed within the remaining time (first memory access in Figure 5.3b). The delay is \(s_{n}^{\text{end}} - t_r + s_{n}^{\text{start}}\) if the core’s bus access cannot be completed within its time slot (third memory access in Figure 5.3b). The maximum bus delay occurs when \(t_r = s_{n}^{\text{end}} - t_m + 1\). TDMA arbitration ensures that all the bus delays experienced by a core only depend on the bus schedule; bus delays are unaffected by the actions of other cores. This gives TDMA a time-composability property [225] that makes its use popular in time predictable systems.

### 5.4 Reachability Technique

The ForeCast analyzer computes the WCRT of a ForeC program by explicit path-enumeration with the reachability technique. Recall from Chapter 4 that the ForeC compiler generates concurrent threads that correspond to those in the original ForeC program. In each tick, the threads execute their respective local ticks. All the possible ticks can be found by traversing the program’s CCFG using ForeC semantics, i.e., exploring all the possible program executions. Due to the program’s control-flow, there may be some local ticks that can never execute in the same tick. Finding the feasible combinations of local ticks is known as the tick-alignment problem [248, 246]. For Figure 5.4a, Figure 5.4b lists all the possible combinations.
5.4 Reachability Technique

(a) Example CCFG.

(b) All possible combinations of local ticks for threads \( t_A, t_C, \) and \( t_D \).

\[
\begin{align*}
\{n3, n7, n10\} & \quad \{n3, n7, n12\} \\
\{n3, n9, n10\} & \quad \{n3, n9, n12\} \\
\{n5, n7, n10\} & \quad \{n5, n7, n12\} \\
\{n5, n9, n10\} & \quad \{n5, n9, n12\}
\end{align*}
\]

(c) Feasible combinations of local ticks for threads \( t_A, t_C, \) and \( t_D \).

\[
\begin{align*}
\{n3, n7, n10\} & \quad \{n5, n9, n12\} \\
\{\max(n3, n5), \max(n7, n9), \max(n10, n12)\}
\end{align*}
\]

(d) Maximum combination of local ticks for threads \( t_A, t_C, \) and \( t_D \).

(e) Feasible ticks.

Figure 5.4: Using reachability to find ticks in a CCFG.

of local ticks among threads \( t_A, t_C, \) and \( t_D \), while Figure 5.4c lists only the feasible combinations. Note that infeasible paths in each local tick can exist, e.g., from programmer written code. Thus, for precise WCRT computation, the analysis must prune away infeasible tick-alignments and paths.

The intuition for using reachability is based on the observation that the CCFG traversal can be guided by contextual information along each path. The contextual information is used to automatically prune away infeasible tick-alignments and paths. Figure 5.4e shows the reachable ticks of Figure 5.4a. Each tick in the figure is a feasible combination of local ticks, annotated with the traversed CCFG nodes. Reachability begins from the start of the CCFG. Whenever a condition node (e.g., \( n1 \)) is reached, the tick is duplicated (e.g., ticks 1a and 1b in Figure 5.4e) to explore both outgoing edges. Tick 1a explores the conditional branch that is \textit{true}, which leads to the last CCFG node (\( n15 \)) and the tick ends. Tick 1b explores the conditional branch that is \textit{false}, which leads to a fork node. Whenever a fork node (e.g., \( n2 \) or \( n6 \)) is reached, each child thread is traversed. Tick 1b ends when
the pause nodes (n4, n8, and n11) of each child thread is reached. The program’s
next tick can only continue from tick 1b. Before the next tick can start, the
triggering of the enclosing strong \texttt{abort} is explored. Thus, tick 2a explores the
possibility that it is triggered, while tick 2b explores the possibility that it is not
triggered. Whenever a join node (e.g., n13 or n14) is reached by all its child threads,
the traversal continues from the join node to the parent thread. If the CCFG
contains loops, then some previously visited ticks will be reached. Previously
visited ticks are ignored because their WCET have already been computed. This
guarantees that reachability will reach a fix-point and terminate.

Reachability can be used to traverse the CCFG at different levels of granularity
(abstraction). For example, a \textit{fine-grained} approach would find all the feasible
combinations of local ticks (e.g., Figure 5.4c). The WCRT would be computed at
a higher level of precision, but require a longer analysis time. A \textit{coarse-grained}
approach would ignore the tick-alignment problem and reduce each thread in the
CCFG into a single worst-case local tick (e.g., Figure 5.4d). The WCRT would be
computed at a lower level of precision from the reduced CCFG, but require shorter
analysis time. The following sections describe these two approaches.

\subsection{Fine-Grained Reachability}

Figure 5.4e is an example of fine-grained reachability, where the threads are ex-
plicitly explored together. To compute the WCRT of the program, we need to
compute the WCET of each tick. Figure 5.5c is a trace showing how the WCET
of tick 1b, from Figure 5.4e, is computed. We assume that the total order shown
in Figure 5.5a and allocation shown in Figure 5.5b are used to statically schedule
the threads over two cores. The execution times of the cores are tracked by their
own \textit{integer counters} (initially 0) and are shown as traces (labeled Core 1 and
Core 2) starting from the left of Figure 5.5c. When the CCFG is traversed, the
counters are incremented by the execution times of the assembly instructions from
the core’s allocated threads and synchronization routines. Each segment in the
trace corresponds to the execution time of a thread or routine. Having a counter
for each core allows us to handle the overlaps in thread and routine execution
times due to parallel execution. To respect the thread scheduling order of each
core, the threads in the CCFG are traversed according to their total order. The
timing model of the hardware is used to compute the execution time of each as-
sembly instruction. When a global memory access is encountered, the bus delay
is computed using equation (5.3). The core’s execution time is taken as the bus
request time, from which $t_r$ is derived. To keep Figure 5.5c concise, these global
memory accesses and associated bus delays are not drawn. We assume that global variables are used to implement the non-blocking sends and blocking receives of the synchronization routines. Hence, global memory accesses are needed to complete each synchronization. The horizontal line between the cores represents the shared TDMA bus that is used to access global memory. When a routine sends data, an arrow is drawn from the routine to global memory. When a routine receives data, an arrow is drawn from global memory to the routine. The blocking time needed to receive data is drawn as a patterned segment in the trace.

When the traversal of a thread reaches a context-switching point (e.g., a pause, fork, or join node), we can deduce the thread’s execution state (e.g., paused, forked, or terminated, respectively). Static scheduling (Chapter 4.2) allows us to determine the synchronization routines needed at each context-switching point. For example, when a thread forks, the cores synchronize by using the mFork and sFork routines. When a nesting of child threads have executed, the cores synchronize by using the mJoin and sJoin routines. For convenience, Table 5.1 summarizes the routines that the compiler generates (Chapter 4). Each type of routine is generated from a parameterized template. For example, the mFork routine is parameterized by the threads, par identifier, and any enclosing abort statements. By compiling with optimizations disabled, the resulting assembly code is also structured and parameterized. The ForeCast analyzer takes advantage of this property and has internal timing models of each type of routine. The models are parameterized to allow accurate execution times to be computed for all possible scheduling scenarios.

Apart from the mAbort and sAbort routines, all other routines use non-blocking
**Table 5.1: Summary of the synchronization routines.** (Reproduction of Table 4.1 for the reader’s convenience.)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mFork</td>
<td>Uses a non-blocking send to notify the slave cores whether or not the parent thread has forked.</td>
</tr>
<tr>
<td>sFork</td>
<td>Blocks until it receives whether or not the parent thread has forked.</td>
</tr>
<tr>
<td>mJoin</td>
<td>Blocks until it receives whether the child threads on other cores have terminated. Then, it notifies the slave cores whether the parent thread has resumed.</td>
</tr>
<tr>
<td>sJoin</td>
<td>Uses a non-blocking send to notify the master core whether or not its child threads have terminated. Then, it blocks until it receives whether the parent thread has resumed.</td>
</tr>
<tr>
<td>mSync</td>
<td>Synchronizes with all the cores, performs the housekeeping tasks, and then synchronizes with all the cores again to start the next global tick.</td>
</tr>
<tr>
<td>sSync</td>
<td>Synchronizes with all the cores and waits for the next synchronization to start the next global tick.</td>
</tr>
<tr>
<td>mAbort and sAbort</td>
<td>Evaluates the preemption condition of an abort.</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the synchronization routines. (Reproduction of Table 4.1 for the reader’s convenience.)

sends and blocking receives on data to synchronize the cores. The blocking times can be computed precisely by knowing the latest time that the data is sent. For example, the WCET computation of tick 1b in Figure 5.5c begins with thread main. When its fork node (n2) is reached, the execution times of mFork1 and sFork1 are computed. Both routines execute some instructions before proceeding to send or receive data. The remaining execution time of mFork1 can be computed because its send is non-blocking. The blocking time for sFork1 ends at the sending time of mFork1, with consideration for any bus delays. When thread tB reaches its fork node (n6), the data sent by mFork2 occurs before sFork2 starts blocking. Hence, the blocking time of sFork2 is zero, with consideration for any bus delays. The same principles are used to resolve the blocking times of the mJoin, sJoin, mSync, and sSync routines. When reachability reaches the end of the tick, the mSync and sSync routines are analyzed on their respective cores. Finally, the execution time of the tick is computed as the maximum execution time computed among all the cores. For example, the WCET of the tick shown in Figure 5.5c is $\max\{\text{Core 1, Core 2}\}$.

The worst-case time complexity of fine-grained reachability is $O(\tau_1 \times \cdots \times \tau_T \times I \times N \times P \times S \times T)$, where:

- $\tau_1 \times \cdots \times \tau_T$ is the maximum number of unique global ticks that must be explored, where $\tau_t$ is the number of local ticks in thread $t$.

- $I$ is the maximum number of CCFG nodes that must be traversed per global tick.
5.4 Reachability Technique

- $N \times P$ is the maximum number of synchronization routines per global tick and depends on the number of cores $N$ and $\text{par}$ statements $P$.

- $S \times T$ is the maximum number of copies of shared variables that have to be combined per synchronization routine and depends on the number of shared variables $S$ and the number of threads $T$.

Note that it takes constant time to calculate the TDMA bus delay experienced by a memory instruction. A recent study [246] suggests that the complexity of the tick alignment problem is at least NP-hard.

5.4.2 Coarse-Grained Reachability

The goal of coarse-grained reachability is to be less complex than fine-grained reachability, which grows exponentially with the number of threads. This is achieved by implicitly exploring the threads together, similar to the approach of Boldt et al. [60] for single-cores. The strategy is to reduce each thread in the CCFG into a single worst-case local tick. Computing the program’s WCRT from the reduced CCFG would be much faster, but possibly less precise. Figure 5.6 is a reduced CCFG of Figure 5.4a. For each thread, fine-grained reachability is used to compute the WCET of all its reachable local ticks. The entire thread is replaced by a single computation node with an execution time equal to the computed WCET.
If a parent thread forks child threads, then a computation node is placed before the fork node and after the join node, each with an execution time equal to the computed WCET. This can be seen in Figure 5.6 for threads main and tB. This is needed because information is lost about whether the child threads are forked at the start or end of the parent thread’s local tick. Note that the reduced CCFG only has one reachable tick because each thread is a single instantaneous path. Thus, coarse-grained reachability assumes that the worst-case local ticks of all threads can execute in the same tick (e.g., Figure 5.4d compared to Figure 5.4c). In other words, the worst-case local ticks of all threads are assumed to always align. Using the reduced CCFG, the program’s WCRT is computed using fine-grained reachability (Chapter 5.4.1). The computation uses the worst-case execution times of the synchronization routines by considering the possible scheduling scenarios.

The worst-case time complexity of coarse-grained reachability is $O((\tau_1 + \cdots + \tau_T) \times I \times N \times P \times S \times T)$, where:

- $\tau_1 + \cdots + \tau_T$ is the maximum number of unique local ticks that must be explored, where $\tau_t$ is the number of local ticks in thread $t$.
- $I$ is the maximum number of CCFG nodes that must be traversed per local tick.
- $N \times P$ is the maximum number of synchronization routines per global tick and depends on the number of cores $N$ and par statements $P$.
- $S \times T$ is the maximum number of copies of shared variables that have to be combined per synchronization routine and depends on the number of shared variables $S$ and the number of threads $T$.

Note that it takes constant time to calculate the TDMA bus delay experienced by a memory instruction.

### 5.5 Optimizations

The ForeCast analyzer does not currently prune infeasible paths due to conditional branches [252]. Thus, ForeCast explores all combinations of branches, leading to an over-approximation of the feasible ticks. That is, some infeasible ticks are considered to be feasible. However, this allows us to dramatically reduce the complexity of the CCFG [252] and improve the performance of ForeCast without losing anymore precision. We illustrate two CCFG optimizations on the sub-graph shown in Figure 5.7a. The cost to execute a node is placed on its incoming edge.
5.6 Discussion

5.5.1 Merging Computation and Condition Nodes

The outgoing edge of a computation or condition node is merged with the outgoing edges of its destination node. When the edges are merged, their costs are added together. This reduces the number of CCFG nodes that have to be traversed. Applying this optimization to Figure 5.7a produces Figure 5.7b.

5.5.2 Merging Edges

When there are multiple edges between two nodes (Figure 5.7b), reachability explores each of the edges. We observe that the tick with the longest execution time always has the edge with the longest execution time. Thus, we only keep the edge with the longest execution time (Figure 5.7c). This reduces the program’s reachable state-space by reducing the number of paths in the CCFG.

5.6 Discussion

This chapter has presented the ForeCast static timing analyzer for parallel synchronous programs, specifically ForeC. ForeCast is implemented using Flex and Bison with 34 C++ supporting classes to scan and parse the Xilinx MicroBlaze binary files. The front-end uses the GNU Objdump tool to disassemble the binary file into assembly code. The assembly code is checked and context information is extracted from the assembly comments to help generate an intermediate format of the program’s CCFG. The XML file generated by the ForeC compiler is used to set the processor architecture, thread hierarchy, preemption scopes, thread-to-core allocation, and shared variables for the analysis. The middle-end annotates all the assembly instructions with their statically known execution times. If re-
quested by the user, the complexity of the CCFG is reduced by merging all possible
nodes and edges. Errors are produced when potentially instantaneous loops are
detected in the CCFG, which prevent reachability from terminating. The back-
end performs reachability on the CCFG and resolves the synchronization costs.
The computed WCRT is reported along with its execution trace. ForeCast can be
improved by determining with greater precision if a loop is instantaneous and if
a dereferenced variable is located in local or global memory, and by incorporating
value analysis to prune infeasible paths. ForeCast uses the reachability technique
to explore, in a fine-grained or coarse-grained manner, the reachable ticks in the
program’s CCFG. A recent study [246] has shown that the complexity of the tick
alignment problem is at least NP-hard. Shared bus delays are easily computed
during reachability by using a time predictable arbitration such as TDMA. The
ForeC compiler facilitates the analysis process by generating synchronization rou-
tines that the ForeCast analyzer could accurately model. By leveraging statically
known information about thread scheduling, ForeCast resolves the routines that
will be executed at each context-switching point and their required blocking times.
The trade-off is that ForeCast is very sensitive to any changes to the structure of
the scheduling routines. The cycle-accurate timing models of the routines must
be updated. ForeCast is also very sensitive to changes to the output of the GNU
Objdump tool. The Lex syntax and Yacc grammar must be updated. To mit-
igate the state-space explosion problem, optimizations to reduce the CCFG and
the coarse-grained reachability approach were presented.

A range of implicit and explicit path-enumeration approaches exist for the
static WCRT analysis of synchronous programs on single-cores. These approaches
include ILP [248, 247, 249], model checking [250, 244, 251], and reachability [60,
252]. These approaches cannot be applied directly to the analysis of multi-cores
because of the nuances of parallel execution and shared resources. Ju et al. [47]
offer the only alternative static WCRT analysis approach for multi-cores. Their
compilation approach generates sequential code for each core, which does not re-
quire thread scheduling. Signal resolution nodes are used to resolve signal statuses
at runtime among the cores. Tick-alignment information is extracted from the gen-
erated code and used to prune infeasible tick-alignments during their reachability-
based analysis. The blocking times due to signal resolution are computed from
the maximum times that the participating cores take to reach their respective sig-
nal resolution nodes. In contrast, ForeCast analyzes parallel programs that are
compiled into threads that fork at runtime across the cores. The synchronization
routines needed to manage the scheduling of threads are analyzed in detail.
5.6 Discussion

Although the static WCRT analysis of synchronous programs on multi-cores has received limited attention, more studies exist on the static WCET analysis of multi-threaded programs on multi-cores [224, 225, 227, 229]. These works have focused on analyzing the low-level hardware features without much consideration for the high-level software behavior. In contrast, ForeCast focuses more on analyzing software-based thread scheduling and synchronization in a scalable manner. Afterall, these are important features found in the practical implementation of parallel programs. Gustavsson et al. [229] use model checking to analyze lock acquisitions by different threads. A timed automata is created for each lock to model the lock acquisitions. However, the use of timed automata leads quickly to a state-space explosion. Moreover, they do not consider programs with nested threads or cores that context-switch between threads. Ozaktas et al. [231] use ILP to analyze the execution time of threads that use synchronization constructs. However, they assume that all threads execute the same code. Moreover, they do not consider the effects of thread scheduling on the resulting WCET. In contrast to these approaches, ForeCast can analyze nested threads, cores that context-switch between threads, and the effects of thread scheduling on the execution time.
This chapter quantitatively assesses ForeC’s parallel execution performance and amenability to static timing analysis on a mixture of data and control dominated benchmark programs. ForeC’s execution performance is compared with that of Esterel, a widely used synchronous language for concurrent safety-critical systems, and that of OpenMP, a popular desktop solution for parallel programming. The precision of the ForeCast static timing analyzer is assessed by comparing the computed WCRT of the benchmark programs with their measured WCRT.

6.1 Benchmark Programs

This section describes the benchmark programs used in the evaluations. The following programs have real-time constraints:

FlyByWire is based on the real-time UAV benchmark called PapaBench [15] that contains tasks with prescribed execution frequencies of 40Hz, 20Hz, 10Hz, and 4Hz. FlyByWire is a control dominated program with several tasks managing the UAV’s motors, navigation, timer, and operation mode.

FmRadio [261] is based on the GNU Radio Package [262] for implementing software radios that typically sample at a frequency of 32kHz. FmRadio transforms a fixed stream of radio signals into audio. The history of the radio signals is used to determine how the remaining stream of signals should be transformed. FmRadio is data orientated.

802.11a [261] is production code from Nokia that tests various signal processing algorithms needed to decode 802.11a data transmissions at a rate of 54Mbit/s. 802.11a has complex data and control dominated computations.
The following programs do not have real-time constraints, but are included because they use parallel programming patterns and have data and control dependencies that appear in real-time applications:

**Life** simulates Conway’s Game of Life [263] for a fixed number of iterations and a given grid of cells. In each iteration of the simulation, the outcome of each cell can be computed independently. **Life** has a good mixture of data and control dominated computations.

**Lzss** uses the Lempel-Ziv-Storer-Szymanski (LZSS [264]) algorithm to compress a fixed amount of text. Multiple sliding windows are used to search different parts of the text for repeated segments that can be compressed. **Lzss** has a good mixture of data and control dominated computations.

**Mandelbrot** computes the Mandelbrot set for a square region of the complex number plane. The Mandelbrot set for each point in the region can be computed independently, making **Mandelbrot** a data-parallel program.

**MatrixMultiply** computes the matrix multiplication of two equally sized square matrices. Each element in the resulting matrix can be computed independently, making **MatrixMultiply** is a data-parallel program.

### 6.2 Performance Evaluation

The performance of ForeC is evaluated against that of Esterel, a traditional synchronous language, and that of OpenMP, a general purpose parallel programming extension to C. The following benchmark programs were selected for the evaluation: **FlyByWire**, **FmRadio**, **Life**, **Lzss**, **Mandelbrot**, and **MatrixMultiply**. We created C (sequential), ForeC, Esterel, and OpenMP versions of each benchmark program. Each version was handcrafted for best performance and Table 6.1 lists the design patterns used to parallelize the ForeC and OpenMP versions. The Esterel versions were limited to the fork-join design pattern. We use **Speedup** as the performance metric to compare ForeC, Esterel, and OpenMP:

\[
\text{Speedup}(P) = \frac{\text{Execution time of the sequential C version}}{\text{Execution time of } P}
\]

where \( P \) is either the ForeC, Esterel, or OpenMP version of the benchmark program being tested. Hence, the speedups of ForeC, Esterel, and OpenMP are always with respect to the execution time of the C version. A higher speedup is better.
6.2 Performance Evaluation

Table 6.1: Parallel design patterns exploited.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ForeC</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlyByWire</td>
<td>Nested fork-join and early-termination</td>
<td>Inner loop data-parallelism</td>
</tr>
<tr>
<td>FmRadio</td>
<td>Pipeline and fork-join</td>
<td>Inner loop data-parallelism</td>
</tr>
<tr>
<td>Life</td>
<td>Nested fork-join</td>
<td>Outer and inner loop data-parallelism</td>
</tr>
<tr>
<td>Lzss</td>
<td>Fork-join</td>
<td>Outer and inner loop data-parallelism</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>Fork-join</td>
<td>Outer loop data-parallelism</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>Fork-join</td>
<td>Outer loop data-parallelism</td>
</tr>
</tbody>
</table>

Xilinx MicroBlaze, 4 physical cores, three-stage pipeline, no speculative features (no branch prediction, caches, or out-of-order execution), 16 KB private data and instruction scratchpads on each core (1 cycle access time), 64 KB global memory (5 cycle access time), TDMA shared bus (5 cycle time slots per core), Benchmarks compiled with GCC-4.1.2 -O0.

Figure 6.1: MicroBlaze multi-core configuration.

6.2.1 Comparison with Esterel

The approaches by Yuan et al. [48, 130] for parallelizing the execution of Esterel programs has been shown to perform well on an Intel multi-core and on a (simulated) Xilinx MicroBlaze multi-core. However, only compiler support is available for Yuan et al.’s dynamic scheduling approach on MicroBlaze multi-core. Thus, we evaluated ForeC against Esterel on a MicroBlaze multi-core and used the dynamic scheduling approach to parallelize the Esterel programs. The static scheduling approach presented in Chapter 4 was used to parallelize the ForeC programs. Yuan et al.’s dynamic scheduling approach uses a special hardware FIFO queue to allocate the threads to the cores. Each core retrieves a thread from the queue and executes it until it terminates or reaches a context-switching point for resolving signal statuses. A core makes a context-switch by adding the executing thread back to the queue and retrieving a different thread from the queue. Threads are added to the queue when they are forked by other threads. Threads are removed from the queue when they terminate. All the cores can access the FIFO queue in parallel and each access takes two clock cycles to complete. For benchmarking, the MicroBlaze multi-core simulator described in Chapter 1.4.1 was extended with a hardware queue to support the dynamic scheduling. The configuration of the simulator is shown in Figure 6.1.

Table 6.2 shows the implementation details of the ForeC and Esterel versions.
Table 6.2: ForeC versus Esterel benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ForeC</td>
<td>Esterel</td>
</tr>
<tr>
<td>Life</td>
<td>212</td>
<td>139+111</td>
</tr>
<tr>
<td>Lzss</td>
<td>485</td>
<td>42+421</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>381</td>
<td>220+337</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>162</td>
<td>51+53</td>
</tr>
</tbody>
</table>

of the benchmark programs. Esterel is suited for specifying control concurrency, but is weak on specifying data dominated computations. Esterel allows data computations to be delegated to external host functions, defined in a host language such as C. Hence, for the “Lines of Code” column in Table 6.2, the first number is the lines of Esterel code and the second number is the lines of host C-code (excluding header files). The “Number of Threads” column specifies the total number of threads forked by the programs and, in brackets, the total number of threads that can execute together in parallel. The benchmark programs were compiled for bare-metal execution and did not need operating system support. Yuan et al.’s compilation approach \cite{48,130} uses an intermediate format called GReph Code (GRC) \cite{118}, which transforms the program into an acyclic execution graph. The GRC helps schedule the resolution of signals and executing the GRC from the top to the bottom corresponds to one tick of the program. To decide which GRC states need to be executed during each tick, a set of internal variables are updated as the GRC is executed. Compared to GRC, for most programs, the ForeC compiler (see Chapter 4) can generate more efficient code that require less context-switching. The same input vector is given to the ForeC and Esterel versions of a benchmark program to ensure the same computations are performed. When a program terminates, the simulator returns the execution time in clock cycles.

Figure 6.2a shows the speedups achieved by ForeC and Esterel when the benchmark programs were executed on four cores. Apart from MatrixMultiply, ForeC shows superior performance compared to Esterel, even though Esterel uses dynamic scheduling with hardware acceleration. This is because the runtime resolution for instantaneous signal communication in Esterel can lead to significant overheads. All possible signal emitters must execute before any signal consumers can execute. In comparison, shared variables in ForeC only need to be resolved at the end of each tick. The significance of the overhead is evident in the Mandelbrot results, where the Esterel version has 24 unique signals and only achieves a speedup of 1.2× on four cores. In fact, when Mandelbrot was executed on one core, Esterel’s execution time was already 58% longer than the C version. ForeC’s execution time
Figure 6.2: Average speedup results for ForeC, Esterel, and OpenMP on four cores normalized to sequential runtime.
Intel Core-i5 3570 at 3.4 Ghz, 4 physical cores, Hyper-Threading disabled, Turbo Boost disabled, SpeedStep disabled, 3 MB L3 data-cache, Linux 3.6, 8 GB of RAM, Benchmarks compiled with GCC-4.8 -O2.

Figure 6.3: Intel multi-core configuration.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ForeC</td>
<td>OpenMP</td>
</tr>
<tr>
<td>FlyByWire</td>
<td>241</td>
<td>227</td>
</tr>
<tr>
<td>FmRadio</td>
<td>481</td>
<td>382</td>
</tr>
<tr>
<td>Life</td>
<td>325</td>
<td>268</td>
</tr>
<tr>
<td>Lzss</td>
<td>593</td>
<td>552</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>111</td>
<td>89</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>156</td>
<td>121</td>
</tr>
</tbody>
</table>

Table 6.3: ForeC versus OpenMP benchmarks.

was only 0.2% longer than the C version. For MatrixMultiply, the fork-join pattern was used by ForeC and Esterel. Because of minimal data dependencies in MatrixMultiply, combine functions were not needed in the ForeC version and signals were not needed in the Esterel version. Thus, the scheduling overheads for the ForeC and Esterel versions were minimal, resulting in close speedup values.

6.2.2 Comparison with OpenMP

Table 6.3 shows the implementation details of the ForeC and OpenMP versions of the benchmark programs. Intel VTune Amplifier XE 2013 [265] software was used during the development and testing of the parallelized benchmarks. The software was useful in providing insight into the regions of code which are the most time consuming and therefore top candidates for parallelization. It was also used to calculate the total runtimes and show the number of cores being used during execution of each benchmark. Figure 6.3 shows the specifications of the desktop computer on which all testing was carried.

Figure 6.2b shows the speedups achieved by ForeC and OpenMP when the benchmark programs were executed over four cores. The speedups were averaged over 200 executions of each program to take into account the potential effects of long term use, e.g., filling and flushing of the cache, and background tasks on the computer. ForeC and OpenMP both achieved a speedup factor of between two and four over four cores. However, from the results, it is clear that ForeC produces a higher speedup factor in most cases, barring two exceptions. First, OpenMP was more suited and delivered a much higher speedup factor in FlyByWire. Second, for Mandelbrot, the OpenMP version could not utilize all four available cores,
possibly due to data dependencies that could not be resolved at runtime.

It should be mentioned that dynamic and static thread scheduling pragmas were used in the OpenMP programs. Static scheduling was used in benchmarks (e.g., FlyByWire) when we could determine at compile time the so called chunk size (the amount of work and number of loop iterations that each thread needs to perform). Dynamic scheduling was used in benchmarks (e.g., MatrixMultiply and Mandelbrot) when the chunk size of each thread could not be made equal or could not be determined at compile time. For dynamic scheduling, the chunk size of each thread is determined by the OpenMP runtime. Using dynamic scheduling does introduce slight overheads, especially thread locking, but these overheads should be amortized over the overall run of the benchmarks. This OpenMP scheduling approach is in stark contrast to the ForeC approach, where all work scheduling is static and determined automatically by the ForeC compiler, whereas in OpenMP all work scheduling is the programmer’s burden.

6.3 Discussion

The first half of this chapter assessed the performance of ForeC. On an embedded multi-core, most of the statically scheduled ForeC programs performed better than Yuan et al.’s [130] dynamically scheduled Esterel programs. This is because it is easier to extract parallelism from ForeC programs, largely thanks to its shared variable semantics (see Chapter 3.1.3). Serializing Esterel programs into GRC can obfuscate the parallelism and the need to update internal state variables can add unnecessary overhead. Runtime resolution is also needed to resolve Esterel’s instantaneous signal communication. Ju et al. [47] provide a multi-core static scheduling approach for Esterel. However, we cannot compare with that work because speedup results for multi-core execution were not reported.

On a desktop multi-core, ForeC’s static scheduling approach proved to be competitive against OpenMP, a dynamic runtime solution. These are encouraging results for the use of ForeC to develop high performing parallel programs. Moreover, determinism is enforced by ForeC’s formal semantics, not by a particular runtime environment. There is much scope to improve the ForeC compiler to generate more efficient code. For example, thread allocations could be refined automatically by feeding the WCRT results of the ForeCast analyzer into the ForeC compiler until the WCRT cannot be reduced. The remainder of this chapter assesses the ForeCast analyzer and the timing predictability of ForeC programs.
Xilinx MicroBlaze, three-stage pipeline, no speculative features (no branch prediction, caches, or out-of-order execution), 8 KB private data and instruction scratchpads on each core (1 cycle access time), 32 KB global memory (5 cycle access time), TDMA shared bus (5 cycle time slots per core and, thus, a $5 \times \text{number of cores}$ cycles long bus schedule), Benchmarks compiled with MB-GCC-4.1.2 -O0 and decompiled with MB-OBJDUMP-4.1.2.

Figure 6.4: MicroBlaze multi-core configuration.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of ForeC</th>
<th>Number of Threads</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlyByWire</td>
<td>204</td>
<td>9 (7)</td>
<td>2</td>
</tr>
<tr>
<td>FmRadio</td>
<td>808</td>
<td>13 (4)</td>
<td>1</td>
</tr>
<tr>
<td>Life</td>
<td>1020</td>
<td>11 (8)</td>
<td>2</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>232</td>
<td>9 (8)</td>
<td>1</td>
</tr>
<tr>
<td>802.11a</td>
<td>2147</td>
<td>26 (10)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.4: ForeC benchmarks.

6.4 Static Timing Analysis

This section assesses the precision and performance of ForeCast’s proposed reachability approach (Chapter 5.4). ForeCast supports fine-grained reachability (FGR) and coarse-grained reachability (CGR). ForeCast is implemented in C++ and targets programs compiled for execution on the MicroBlaze multi-core simulator, presented in Chapter 1.4.1. The configuration of the simulator is shown in Figure 6.4. ForeCast was executed on a 2.20 GHz Intel Core 2 Duo computer with 3 GB RAM and Linux 2.6.38. We chose the following ForeC benchmark programs: FlyByWire, FmRadio, Life, Lzss, MatrixMultiply, and 802.11a. In Table 6.4, the second column gives the number of lines of ForeC code. The third column specifies the total number of threads forked by the programs and, in brackets, the total number of threads that can execute together in parallel. The fourth column gives the deepest level of thread nesting. Note that earlier versions of the benchmark programs and ForeC compiler were used for this evaluation, compared to the performance evaluation. Hence, the slight discrepancies in program characteristics.

We analyzed each program using FGR and CGR, and recorded the computed WCRTs. The programs were distributed up to their maximum number of parallel threads and the analysis was repeated for each possible distribution. ForeCast reports the program state of the computed WCRT and produces a corresponding timing trace. These useful timing traces were used to (manually) refine the thread distributions of each program. To evaluate the tightness of the computed WCRTs, we executed the programs on the multi-core MicroBlaze simulator (Chapter 1.4.1) for one million global ticks or until the program terminated. Test vectors were
generated to elicit the worst-case program state by studying the program’s control-flow. The simulator reported the execution time of each global tick and the longest was taken as the observed WCRT.

6.4.1 Precision of the Computed WCRT

The observed and computed WCRTs (in clock cycles) of the programs are plotted as line graphs in Figure 6.5. The observed WCRTs demonstrate the benefit of parallel execution, which become shorter when the number of cores increases. However, for 802.11a and FlyByWire, their observed WCRTs become longer when the number of cores continue to increase. This is because of the increasing scheduling overheads and cost of accessing global memory. The observed WCRTs of Life, FmRadio, and MatrixMultiply plateau because it is difficult to distribute the threads such that the workloads of the cores are balanced. For example, although the eight parallel threads of MatrixMultiply have equal workloads, they cannot be distributed evenly over five to seven cores. At least one core must be allocated two threads, while the other cores are allocated one thread, thus, preventing the observed WCRT from decreasing. Only when there are eight cores can the eight threads be executed on their own core. Hence, the sudden decrease in observed WCRT from seven to eight cores. Similar decreases in observed WCRTs can be seen when FmRadio and Life are distributed on four and eight cores, respectively.

Apart from MatrixMultiply, FGR computes much tighter WCRTs than CGR, even when the programs are executed on a single core. CGR has large over-estimations because the assumption that all thread worst-case local ticks occur in the same tick may be invalid. However, the assumption is valid for MatrixMultiply because it completes its execution in one tick. Hence, CGR computes a tight WCRT for MatrixMultiply. For CGR, the amount of over-estimation depends on the program’s structure and thread distribution. For example, the WCETs of sequentially executed threads get summed together. Sequential execution can result from control-flow dependencies, such as the sequencing or nesting of par statements. Threads allocated to the same core will also execute sequentially. Also, the parent thread’s WCET is summed again after each join for a sound WCRT computation. Such program structure contributes towards the high over-estimations for FmRadio and Life. For FGR and CGR, the blocking time for a scheduling routine to receive data is over-estimated by assuming the worst-case scenario. The worst-case is when the the blocking receive checks for new data just before the data is sent over. For FGR and CGR, it is important to determine if a dereferenced variable is located in local or global memory. If the location of a vari-
Figure 6.5: WCRT results for the benchmark programs in clock cycles.
6.4 Static Timing Analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>None</th>
<th>Merge</th>
<th>Merge-b</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlyByWire</td>
<td>50,032</td>
<td>50,032</td>
<td>1,032</td>
</tr>
<tr>
<td>FmRadio</td>
<td>4,296</td>
<td>4,296</td>
<td>36</td>
</tr>
<tr>
<td>Life</td>
<td>2,053</td>
<td>2,053</td>
<td>1,029</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>802.11a</td>
<td>43,695</td>
<td>43,695</td>
<td>515</td>
</tr>
</tbody>
</table>

Table 6.5: Number of reachable ticks.

able cannot be determined, ForeCast assumes that it is located in global memory, leading to possible over-estimation. For FGR and CGR, ForeCast assumes that all copies of shared variables are combined in every tick, even though this may be false, leading to over-estimation.

6.4.2 ForeCast Analysis Time and Memory Usage

For 802.11a, Figure 6.6a plots the analysis time for FGR and CGR without CCFG optimizations. FGR has much higher analysis times than that of CGR because its analysis complexity is much higher. Figure 6.6b plots the analysis time for FGR with CCFG optimizations (Chapter 5.5). Merge denotes the merging of CCFG nodes and Merge-b denotes the merging of CCFG nodes and edges. The average improvement in analysis time was 9.34× for Merge and 342× for Merge-b. For Merge-b, the maximum analysis time was only 6.93 seconds. This dramatic improvement is due to the reduction in the number of reachable ticks (Chapter 5.5.2). Table 6.5 gives the number of reachable ticks for the programs, with optimizations (Merge, Merge-b) and without (None). The gradual increase in analysis time for Merge-b shows its scalability over increasing number of cores.

The peak memory used during the analysis of 802.11a was recorded with the Valgrind Massif tool [266], shown in Figure 6.6c for FGR and Figure 6.6d for CGR. For both approaches, the peak memory increased linearly with additional cores. Only the data structures needed to track the core execution times are replicated when additional cores are considered. For FGR, each core’s execution trace is stored for each reachable tick for debugging purposes. Hence, the peak memory usage occurs when all reachable ticks have been found. The peak memory usage for FGR drops slightly when the Merge optimization is used because less nodes need to be stored in the execution traces. It drops significantly when Merge-b is used because the number of reachable ticks is reduced to 515 (Table 6.5). For CGR, we only store the execution time of each thread’s reachable local ticks. Consequently, the peak memory usage only occurs when creating the reduced CCFG. Figures 6.7–6.10 plot the analysis times and memory usages for the other benchmark programs.
Figure 6.6: Analysis times and memory usage for 802.11a.
6.4 Static Timing Analysis

(a) FGR and CGR analysis time without optimizations.

(b) FGR analysis time with optimizations.

(c) FGR peak memory usage.

(d) CGR peak memory usage.

Figure 6.7: Analysis times and memory usage for FlyByWire.
(a) FGR and CGR analysis time without optimizations.

(b) FGR analysis time with optimizations.

(c) FGR peak memory usage.

(d) CGR peak memory usage.

Figure 6.8: Analysis times and memory usage for Life.
6.4 Static Timing Analysis

![Graphs showing analysis times and memory usage for FmRadio.](image)

(a) FGR and CGR analysis time without optimizations.

(b) FGR analysis time with optimizations.

(c) FGR peak memory usage.

(d) CGR peak memory usage.

Figure 6.9: Analysis times and memory usage for FmRadio.
(a) FGR and CGR analysis time without optimizations.

(b) FGR analysis time with optimizations.

(c) FGR peak memory usage.

(d) CGR peak memory usage.

Figure 6.10: Analysis times and memory usage for MatrixMultiply.

Figure 6.11: 802.11a’s computed and observed WCRT speedups.
6.4.3 WCRT Speedup

Figure 6.5 shows that the observed WCRTs of the programs benefit from multi-core execution. The observed WCRT speedup can be calculated as follows:

\[
\text{Observed WCRT Speedup}(n) = \frac{\text{Observed WCRT for one core}}{\text{Observed WCRT for } n \text{ cores}}
\]

The WCRT speedups were partly achieved by using the WCRT timing traces produced by ForeCast to refine the thread allocations. MatrixMultiply had the greatest speedup of 5.94× on eight cores. Although its eight parallel threads are symmetrical, they cannot be distributed evenly over five to seven cores. Thus, some cores are always allocated with two threads and the WCRT cannot improve. FlyByWire had the least speedup of 2.12× on four cores. This was because the thread workloads could not be balanced over the cores, which prevented the full utilization of the cores. For 802.11a, its WCRT at five cores corresponded to the execution time of one thread which was already allocated to its own core. Thus, the WCRT could not be improved by distributing the remaining threads. The WCRT increases after five cores because of the increasing scheduling overheads and cost of accessing global memory. These costs reduce the benefit of multi-core execution. The computed WCRTs can be used to estimate the WCRT speedups that a program could achieve when executed on a number of cores. The computed WCRT speedup is calculated as follows:

\[
\text{Computed WCRT Speedup}(n) = \frac{\text{Computed WCRT for one core}}{\text{Computed WCRT for } n \text{ cores}}
\]

Using the WCRTs computed by FGR, Figure 6.11 shows the computed WCRT speedups for 802.11a compared to its observed WCRT speedups. The computed and observed WCRT speedups are very similar because FGR computes the WCRTs to a high precision.

6.5 Discussion

The second half of this chapter assessed the time predictability of ForeC. The results for the ForeCast static timing analyzer showed that the WCRTs computed by FGR are far tighter than those computed by CGR. FGR with CCFG optimizations demonstrated a large reduction in analysis time without trading-off precision, although variables were not tracked. In addition, ForeCast provides a timing trace for the program’s WCRT. This feedback allows for an effective approach for de-
sign space exploration. The efficiency of ForeCast enables timing analysis to be applied early on in the design exploration. Ju et al. [47] offer the only other known static WCRT analysis approach for synchronous programs on multi-cores. Unfortunately, we cannot compare with that work because their results are only for a four core system with no precision results reported by them. ForeCast is significantly different from Boldt et al. [60] and Kuo et al. [252] because these works only consider single-core execution and ignores complex software-based synchronization and thread scheduling, shared bus delays, and overlaps in thread execution times.

This chapter has presented benchmark results that evaluated the performance and time predictability of ForeC. Overall, the benchmarking showed that high performing and time predictable CPSs can be designed with ForeC on top of a time-predictable architecture. Developing CPSs with such ease is highly desirable.
Previous chapters have described the new ForeC language and its supporting compiler and static timing analyzer. Static timing analysis is necessary to validate the synchrony hypothesis, that all threads can complete their executions before new inputs arrive. This requires all threads to be hard real-time and precludes the use of threads with soft or non-real-time deadlines. However, in CPSs, it is common to have tasks with hard, soft, or non-real-time deadlines. In this thesis, the criticality [145] of a task is the level of assurance required against its failure to meet its deadlines. Hence, a hard real-time task is more critical than a non-real-time task and are subject to different levels of certification. This chapter introduces the ForeC Mixed-Criticality (ForeMC) framework that relaxes the synchrony hypothesis to allow the specification of mixed-criticality tasks. With respect to timing, these tasks can have hard, soft, and non-real-time deadline requirements. A suitable communication model is proposed to allow tasks of different criticalities to communicate with each other. Finally, we propose a multi-processor scheduling approach that preserves the hard, soft, and non-real-time requirements of the tasks. The ability for soft real-time tasks to meet their deadlines is improved by allowing them to execute during the processors’ slack time. We believe that the ForeMC framework is applicable to a wide range of CPSs, such as automotive systems [168], UAVs [267], and biomedical devices [268].

7.1 Task Model

The traditional synchronous task model consists of a set of periodic tasks $\tau \in \Gamma$ with period $p_\tau$. Each task has an implicit deadline equal to $p_\tau$, i.e., each task must complete their computations before their next release time. Without loss of generality, we assume that tasks do not create new tasks at runtime and are initially released at the same time. In the relaxed synchronous task model of
ForeMC, the programmer assigns a criticality $\zeta_r$ to each task as either life, mission, or non-critical. That is, $\zeta_r \in \{\text{life, mission, non-critical}\}$. Life critical tasks adhere to the synchrony hypothesis and must complete their computation before their next release time (a hard real-time deadline). For example, if a life critical task with period $p_r$ is released at time $r_r$, then its deadline (and next release) is at time $r_r + p_r$. We relax the synchrony hypothesis for mission critical tasks in that bounded deadline misses are tolerated. For example, if a mission critical task misses its deadline of time $r_r + p_r^{\min}$, then it cannot miss a relaxed deadline of time $r_r + p_r^{\max}$, where $p_r^{\min} < p_r^{\max}$. We relax the synchrony hypothesis completely for non-critical tasks by removing the notion of deadlines. For example, if a non-critical task with period $p_r$ is released at time $r_r$ and completes its computation at time $r_r + q$, then its next release is at time $r_r + \max\{p_r, q\}$.

To better understand the execution rates of the tasks, their periods can be converted into frequencies using the equation $f_r = 1/p_r$. The minimum and maximum release frequencies, $f_r^{\min}$ and $f_r^{\max}$, depend on the assigned criticality $\zeta_r$. For life critical tasks, $f_r^{\min} = f_r^{\max}$. For mission critical tasks, $f_r^{\min} < f_r^{\max}$. For non-critical tasks, only $f_r^{\max}$ is specified and is treated as a goal frequency. All specified frequencies must be greater than zero. Any implementation must ensure that all life critical tasks execute at their $f_r^{\max}$ and that mission critical tasks execute within their $f_r^{\min}$ and $f_r^{\max}$. For non-critical tasks, while the implementation tries to meet their $f_r^{\max}$ (goal frequency), no guarantees are provided, i.e., these tasks may execute at a slower frequency compared to their goal frequency. WCET analysis [44] can be used to estimate an upper bound on a task’s maximum computation time $c_r$. For non-critical tasks, $c_r$ is left unspecified.

For the UAV running example described in Chapter 1.2, Figure 7.1 shows the UAV tasks and their assigned criticalities. When assigning a criticality, we assume that life critical tasks must meet hard real-time deadlines, whereas mission critical tasks can improve their quality of service (QoS) by executing more frequently. Recall that the UAV uses the Navigation and Stability tasks to maintain stable flight to its intended destination. Both these tasks are life critical because they are necessary for safe operation. The Avoidance task checks for obstacles around the UAV and the VideoStream task streams an on-board video of the flight back to the operators. Both these tasks are mission critical because high video frame rates and frequent checking of obstacles is desirable, with their lowest tolerable frequency being $10\text{Hz}$. The Logging task logs important flight events and the Sharing task shares information with nearby UAVs. Both these tasks are non-critical because they are auxiliary functions of the UAV.
what level a mission critical task is by taking the ratio of its minimum frequency to its criticality level. Critical tasks correspond to the intermediate levels, while non-critical tasks correspond to the lowest level. Life critical tasks correspond to the highest level. Table 7.1 shows how it can be extended to five levels. Life critical tasks correspond to the highest level, while non-critical tasks correspond to the lowest level. Mission critical tasks correspond to the intermediate levels. The certification standards typically define more than three levels of criticality.

### 7.1.1 Extension to More Levels of Criticality

Certification standards typically define more than three levels of criticality. For example, the DO-178B standard defines five levels, labeled A to E. Although our mixed-criticality task model has only dealt with three levels of criticality, Table 7.1 shows how it can be extended to five levels. Life critical tasks correspond to the highest level A, while non-critical tasks correspond to the lowest level E. Mission critical tasks correspond to the intermediate levels B, C, and D. We can decide what level a mission critical task is by taking the ratio of its minimum frequency to its criticality level.

![Figure 7.1: Tasks of the UAV with criticality levels and frequency bounds.](image-url)

The proposed ForeC language can be extended to support the ForeMC framework. Each task is implemented as a function in ForeC and forked as a thread using the `par` statement. The criticality and release frequencies of each task are specified immediately after the function’s input parameters. For example, Figure 7.2 extends the ForeC code given in Figure 3.3 to include the mixed-criticality tasks of Figure 7.1. The `Navigation` task is defined on line 14 with “life(4Hz)” to specify that it is a life critical task with a constant release frequency of 4Hz. The `Avoidance` task is defined on line 23 with “mission(10Hz, 20Hz)” to specify that it is a mission critical task with a release frequency of between 10 and 20Hz. The `Sharing` task is defined on line 35 with “noncritical(10Hz)” to specify that it is a non-critical task with a goal frequency of 10Hz.
```c
#include <uav.h>

input int pos1, pos2, commIn, proxL, proxR, cam; // Inputs.
output int motors=0, flaps=0, commOut=0, videoOut=0; // Outputs.

typedef struct {
  int id , obst , ...} UAV; // UAV information.

void main(void) {
  shared int obst=0 combine new with min;
  shared int newPos=0, stats =0, log =0;
  par(Navigation(&newPos,&obst),Stability(&newPos,&stats), // Life critical tasks.
      Avoidance(&obst),VideoStream(), // Mission critical tasks.
      Logging(&obst,&stats,&log),Sharing(&log)); // Non-critical tasks.
}

void Navigation(shared int *newPos,shared int *obst) life (4Hz) {
  while (1) {
    *newPos=plan (pos1, obst) ;
    pause;
  }
}

void Stability(shared int *newPos,shared int *stats) life (20Hz) {
  while (1) {
    motors=thrust (pos2, newPos); flaps=direction (pos2, newPos);
    *stats=compress(motors, flaps);
    pause;
  }
}

void Avoidance(shared int *obst) mission (10Hz,20Hz) {
  while (1) {
    par(*obst=find (proxL));{*obst=find (proxR)}; pause;
  }
}

void VideoStream(void) mission (10Hz,25Hz) {
  while (1) {
    videoOut=frame(cam);
    pause;
  }
}

void Logging(shared int *obst,shared int *stats,shared int *log)
  noncritical (10Hz) {
  while (1) {
    *log=compress(obst,stats);
    pause;
  }
}

void Sharing(shared int *log) noncritical (10Hz) {
  while (1) {
    UAV uavs [5]; nearByUAVs(uavs); commsOut=compress(uavs,log);
    pause;
  }
}

int min(int th1, int th2, int pre) {
  if (th1 < th2) {return th1;} else {return th2;}
}

int min(int th1, int th2, int pre) {
  if (th1 < th2) {return th1;} else {return th2;}
}

int min(int th1, int th2, int pre) {
  if (th1 < th2) {return th1;} else {return th2;}
}

Figure 7.2: Example ForeMC program for the UAV running example.

<table>
<thead>
<tr>
<th>Software Level</th>
<th>Failure Condition</th>
<th>ForeMC Criticality</th>
<th>( f_{\text{ratio}} = \frac{f_{\text{min}}}{f_{\text{max}}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>life</td>
<td>( f_{\text{ratio}} = 1 )</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous</td>
<td>mission</td>
<td>( 1 &gt; f_{\text{ratio}} &gt; 0.66 )</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>mission</td>
<td>( 0.66 \geq f_{\text{ratio}} &gt; 0.33 )</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
<td>mission</td>
<td>( 0.33 \geq f_{\text{ratio}} &gt; 0 )</td>
</tr>
<tr>
<td>E</td>
<td>No effect</td>
<td>non-critical</td>
<td>( f_{\text{ratio}} = 0 )</td>
</tr>
</tbody>
</table>

Table 7.1: Extending ForeMC to five criticality levels.
7.2 Delayed Communication Model

to its maximum frequency, i.e., \( f_{\text{ratio}} = f_{\text{min}} / f_{\text{max}} \). A ratio close to 1 implies that task \( \tau \) can only tolerate short deadline misses. A ratio close to 0 implies that task \( \tau \) can tolerate long deadline misses. This extension to five criticality levels can be refined by the certificate authority or application designer, depending on the context. For example, the certificate authority may be more conservative than the application designer [145]. Alternatively, task criticality can also be defined in terms of functional correctness and fault tolerance, e.g., as for Pellizzoni et al. [268] in their pacemaker design. By combining our timing-centric task model with a function-centric task model, more criticality levels may be supported.

7.2 Delayed Communication Model

In this section, we define the communication model for the proposed multi-rate, mixed-criticality, synchronous task model. The synchrony hypothesis requires all communications to complete in the same period that they were started in, i.e., before the tasks are released again. This is shown in Figure 7.3a for two tasks \( A \) and \( B \) that have the same release frequency and are executed on their own processors. The tasks communicate using the variables \( a \) and \( b \). The tasks must be scheduled such that data is always sent before it is received. This data-dependency limits task schedulability and the ability to execute tasks in parallel. One way to relax data-dependencies is to delay the receiving of data by one period. When tasks are released, they receive the data sent from the previous period. For a task’s initial release only, the initialization values of the variables are received. This delayed semantics is illustrated in Figure 7.3b and is supported by some synchronous languages (e.g., ForeC’s shared variable semantics, the pre operator of Esterel [45] and Lustre [116], Prelude’s fby operator [122], and SL’s delayed semantics [269]). In Figure 7.3b, variables \( a \) and \( b \) are assumed to be declared with an initial value. At time \( r \), both tasks receive the initial values \( a_0 \) and \( b_0 \). At time \( r + p \), both tasks receive the values \( a_1 \) and \( b_1 \) sent from the previous period. For the remainder of the paper, we assume that all task communications are delayed to facilitate better parallelism.

7.2.1 Multi-Rate Communication Model

A synchronous program is multi-rate if it has tasks that can be released at different frequencies, i.e., there exist two tasks \( \tau \) and \( \tau' \) where \( f_{\tau} \neq f_{\tau'} \). Consequently, the tasks may need to communicate with each other at different frequencies. A common approach is to oversample data sent from lower frequency tasks and to
Figure 7.3: Task communication models. Tasks depicted as executing on their own processors.
undersample data sent from higher frequency tasks [128, 270, 122]. Figure 7.3c shows oversampling, where task A receives the last value sent from task B multiple times, \{b_0, b_1, b_1, b_2, \ldots\}. Figure 7.3d shows undersampling, where task B only receives the last value sent from task A, \{a_0, a_2, a_4, \ldots\}. The main disadvantage with undersampling is the loss of data, e.g., \{a_1, a_3, \ldots\}, which can be unacceptable when, e.g., commands are sent between tasks. To overcome the problem with undersampling, we propose a simple approach of lossless buffering. The idea of lossless buffers for multi-rate systems is not new. For instance, in Synchronous Data Flow (SDF) [121], lossless buffers with statically bounded sizes are achieved by restricting tasks (typically called nodes or actors) to consume and produce fixed, statically known, number of data items. By contrast, in our methodology, mission-critical tasks can produce dynamically varying number data items; only the minimum and maximum data rates are statically known.

The suggested approach of lossless buffers is intuitive: all data sent from a higher frequency task to a lower frequency task is buffered in a first in, first out (FIFO) buffer. When the lower frequency task is released, it consumes all the data in the buffer and then clears the buffer. This is shown in Figure 7.3e, where the buffer begins with the initial value \(a_0\) for variable \(a\). At time \(r\), the buffer is consumed and cleared. Between the time interval \([r, r + 2p]\), the data sent from task A is buffered. At time \(r + 2p\), task B is released so it consumes and clears the buffer. When task B executes, it has the values \(a_1\) and \(a_2\) for variable \(a\). The programmer is free to use the values in any way they like. For example, if data loss is undesirable, then the task can process the “backlog” of values. As another example, a task may only use the latest value for its computation. In this case, the communication behaves like the undersampled method shown in Figure 7.3d.

We observe two key properties of the proposed lossless buffering approach:

**Observation 1.** The maximum buffer size needed for lossless communication between two life or mission critical tasks is bounded: Let \(\tau\) be a life or mission critical task that communicates to another life or mission critical task \(\tau'\). Let \(\tau\) be the higher frequency task. The maximum buffer size needed for lossless communication is equal to the maximum number of times that \(\tau\) can send data between each release of \(\tau'\). This occurs when \(\tau\) is released at its maximum frequency \((f_{\tau}^{max})\) and when \(\tau'\) is released at its minimum frequency \((f_{\tau'}^{min})\). Thus, the maximum buffer size can be calculated as the ratio of both frequencies:

\[
\text{Maximum buffer size} = \left[ \frac{f_{\tau}^{max}}{f_{\tau'}^{min}} \right]
\]
### Runtime Frequencies of the Sending \( \tau \) and Receiving \( \tau' \) and Communication Model

<table>
<thead>
<tr>
<th>Sending Task Frequency</th>
<th>Receiving Task Frequency</th>
<th>Communication Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\tau = f_{\tau'} ) \ (Sending and receiving task frequencies are identical)</td>
<td></td>
<td>Delayed (Figure 7.3b)</td>
</tr>
<tr>
<td>( f_\tau &lt; f_{\tau'} ) \ (Sending task has lower frequency)</td>
<td></td>
<td>Oversampling (Figure 7.3c)</td>
</tr>
<tr>
<td>( f_\tau &gt; f_{\tau'} ) \ (Sending task has higher frequency)</td>
<td>( \zeta_{\tau'} \in {\text{life, mission}} ), otherwise,</td>
<td>Lossless buffering (Figure 7.3e) if undersampling (Figure 7.3d)</td>
</tr>
</tbody>
</table>

Table 7.2: Mixed-criticality communication model.

**Observation 2.** By using FIFO buffers, the (untimed) sequence of data sent from the higher frequency task is always received in the same sequence by the lower frequency task.

### 7.2.2 Mixed-Criticality Communication Model

We now define the communication model for mixed-criticality tasks, summarized in Table 7.2. All communication between tasks of the same frequency are simply delayed (Figure 7.3b). All communication from lower to higher frequency tasks are oversampled (Figure 7.3c). All communication from higher frequency tasks to lower frequency life and mission critical tasks use lossless buffering (Figure 7.3e). All communication from higher frequency tasks to lower frequency non-critical tasks are undersampled (Figure 7.3d). Lossless buffering is not used because the release time of a non-critical task can be unbounded and an infinite number of data may have to be buffered. This communication model can be implemented as a program library or as a new data type in an existing synchronous language. We continue by describing a possible implementation for ForeC.

Figure 7.2 shows the use of shared variables between the mixed-criticality ForeC tasks, e.g., the shared variable `newPos` on line 8 that is used by the Navigation and Stability tasks. Data is always sent from a task at the end of its local tick and buffered data is always received by a task when it is released. The ForeC compiler can generate code to support the mixed-criticality communication model. For each pair of sending and receiving tasks, each shared variable is transformed into a circular buffer, illustrated in Figure 7.4a and defined in Figure 7.4b. The receiving task reads from the segment that contains the sent data. The segment starts from the index `rFirst` and `rCount` indicates the length of the segment. Meanwhile, the sending task places new data into a new segment. The segment starts from the index `wFirst` and `wCount` indicates the length of the segment so
7.2 Delayed Communication Model

(a) Usage.

typedef struct {
    DataType data[MAX];
    int rFirst, rCount;
    int wFirst, wCount;
} buffer;

(b) C structure.

circular buffer

(c) Syntactic sugar translated into C-code.

Figure 7.4: Circular buffer.
far. By allowing the sending and receiving tasks to occupy separate segments of the circular buffer, the receiving task does not need to immediately consume all of the buffered data when it is released. The trade-off is that the size of the circular buffer (MAX value in Figure 7.4b) must be twice the maximum buffer size needed by both tasks (Observation 1).

We offer some syntactic sugar for accessing the buffered data. Assume that the shared variable $s$ is used for thread communication. Then, $s.count$ returns the number of buffered data that can be read (i.e., $rCount$) and $s[i]$ returns the $i$-th buffered data. The programmer can iterate through the buffered data using a loop. Figure 7.4c shows how the syntactic sugar can be translated into C-code. Lines 1–2 show the translation of $s.count$ into $s.rCount$. Lines 4–5 show the translation of $s[i]$. The function $bound$, defined on line 8, bounds the value of $i$ to $s.rCount$ to ensure that it does not exceed the segment of buffered data. The function $getIndex$, defined on line 17, maps the resulting value of $i$ to its corresponding index in the circular buffer. Line 22 defines the function $send$ that helps place data into the circular buffer. Line 28 defines the function $release$ that helps update the segment that the receiving task can read from when it is released. Line 30 transfers the index and count of the sending task to the receiving task. Lines 33–34 advances the index and resets the number of buffered data for the sending task. In the case that the sending task becomes slower than the receiving task, Line 37 ensures that the receiving task can oversample the last value sent from the sending task.

### 7.3 Scheduling ForeMC for Parallel Execution

In this section, we describe a scheduling approach for the proposed multi-rate, mixed-criticality, synchronous task model. It considers the scheduling of a set of tasks $\tau \in \Gamma$ over a set of processors $n \in N$. The scheduler’s goal is to ensure that the life critical tasks execute at their $f^{max}_{\tau}$ and that the mission critical tasks execute within their $f^{min}_{\tau}$ and $f^{max}_{\tau}$. For non-critical tasks, the scheduler tries to meet their $f^{max}_{\tau}$ but no guarantees are provided, i.e., they may be executed at a slower frequency. Note that the scheduling approach is applicable to other types of parallel architectures, such as multi-core and multi-threaded processors [31, 161]. Task scheduling on multi-processors requires the following decisions to be made:
Allocation of tasks to processors: In the partitioned approach [271], tasks are statically allocated to each processor and do not migrate. In the global approach [271], tasks are dynamically allocated and may migrate across the processors at runtime.

Assignment of task priorities: The scheduler always selects the highest priority tasks to execute on the available processors. Task priorities can be statically assigned. For example, in rate-monotonic [272], tasks with shorter periods have higher priorities. In deadline-monotonic [273], tasks with shorter deadlines have higher priorities. Task priorities can also be dynamically assigned. For example, in earliest deadline first (EDF) [272], tasks with earlier deadlines have higher priorities. Thus, task priorities are recalculated whenever a task is released.

Allowance of task preemptions: Task preemption is the interruption of a task’s execution to allow another task to be scheduled. Task preemption allows the scheduler to execute high priority tasks as soon as they are released. Typically, a task’s execution context is saved when it is preempted to allow its execution to be resumed when it is scheduled for execution at a later time.

A schedulability test is used to decide if a scheduling algorithm can successfully schedule a set of tasks on a set of processors. The test uses task utilization to determine the amount of time that the processors would spend executing the tasks. Task utilization is the ratio of a task’s maximum computation time $c_\tau$ to its period $p_\tau$, i.e., $u_\tau = c_\tau/p_\tau$. Since $f_\tau = 1/p_\tau$, we have $u_\tau = c_\tau f_\tau$. We assume that each processor can only execute one task at a time. Thus, the maximum utilization of each processor is 1. We define additional notions of task utilization:

$$u_{\tau}^{\min} = c_\tau \cdot f_{\tau}^{\min} \tag{7.2}$$
$$u_{\tau}^{\max} = c_\tau \cdot f_{\tau}^{\max} \tag{7.3}$$
$$U_\Gamma^{\text{life}}(\tau) = \sum_{\tau \in \Gamma, \zeta_\tau = \text{life}} u_{\tau}^{\min} \tag{7.4}$$
$$U_\Gamma^{\text{mission}}(\tau) = \sum_{\tau \in \Gamma, \zeta_\tau = \text{mission}} u_{\tau}^{\min} \tag{7.5}$$

where, $u_{\tau}^{\min}$ and $u_{\tau}^{\max}$ is task $\tau$’s minimum and maximum utilization, respectively, and $u_{\tau}^{\min} = u_{\tau}^{\max}$ for life critical tasks only. $U_\Gamma^{\text{life}}$ and $U_\Gamma^{\text{mission}}$ is the total $u_{\tau}^{\min}$ of all the life and mission critical tasks, respectively, in $\Gamma$. 
Definition 6. (Schedulability) A task set $\tau \in \Gamma$ is schedulable over a set of (homogeneous) processors $n \in N$ if:

$$\forall n \in N : U_{\Gamma_n}(life) + U_{\Gamma_n}^{\text{min}}(mission) \leq 1$$

where $\Gamma_n \subseteq \Gamma$ is the set of tasks allocated to processor $n$.

The schedulability condition represented by equation (7.6) only considers life and mission critical tasks. No execution guarantees are made for non-critical tasks. Note that task schedulability is impacted by pessimistic estimates for task computation times and may lead to the over-provisioning of resources.

The proposed scheduling algorithm uses partitioned, static priority, preemptive scheduling to meet the $f^{\text{min}}$ of the life and mission critical tasks. If the actual computation times of the tasks are less than their estimated computation time, then slack will appear on the processors. During this slack, global, dynamic priority, preemptive scheduling is used to try and meet the $f^{\text{max}}$ of the non-critical tasks and to improve the mission critical tasks towards their $f^{\text{max}}$. The following sections describe some existing scheduling approaches for synchronous tasks, followed a detailed explanation of the proposed approach.

7.3.1 Scheduling Approaches

Traditional multi-rate, synchronous tasks can be scheduled online [43, 115, 122] or offline [115, 154]. To guarantee schedulability, the tasks are scheduled for their maximum computation time $c_\tau$. In online scheduling, the multi-rate tasks are scheduled using the rate-monotonic or EDF algorithms. In offline scheduling, the tasks are partitioned over the processors and allocated a fixed amount of time to execute. A static schedule is created such that it can be repeated indefinitely to meet the timing constraints of all tasks. The length of the schedule (makespan) is usually the hyper period or base period of all the task periods. The hyper and base periods are computed as the least common multiple (LCM) and greatest common divisor (GCD) of all task periods, respectively. A side-effect of this is that the task periods must be rational numbers in order to compute the LCM or GCD. Figure 7.5 compares the hyper and base period approaches for scheduling the same multi-rate tasks, $C$ and $D$, on separate processors. Their release frequencies, corresponding periods, maximum computation times (in milliseconds), and utilizations are:

<table>
<thead>
<tr>
<th>Task</th>
<th>Frequency ($f$)</th>
<th>Period ($p$)</th>
<th>Computation Time ($c$)</th>
<th>Utilization ($u$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>$2Hz$</td>
<td>$500ms$</td>
<td>$250ms$</td>
<td>$0.5$</td>
</tr>
<tr>
<td>$D$</td>
<td>$5Hz$</td>
<td>$200ms$</td>
<td>$100ms$</td>
<td>$0.5$</td>
</tr>
</tbody>
</table>
7.3 Scheduling ForeMC for Parallel Execution

7.3.2 Hyper and Base Period Approaches

The hyper period approach \cite{115, 154} constructs a schedule for the shortest time interval in which all tasks can meet their timing constraints. In Figure 7.5a, it can be verified that tasks C and D can continue to execute at $2\text{Hz}$ and $5\text{Hz}$, respectively, when the schedule is repeated indefinitely. However, very long schedules are generated when the hyper period is much larger than the task periods. Longer schedules typically require more memory to store. The base period approach \cite{115} creates shorter schedules by allocating a portion $t_{\tau}$ of the task’s maximum computation time $c_{\tau}$ over each base period. The number of base periods that can be used to execute a task is computed as $p_{\tau}/p_b$, where $p_b$ is the duration of the base period. Hence, $t_{\tau} = \frac{c_{\tau}}{p_{\tau}/p_b}$ or $p_b \cdot u_{\tau}$. The release of a task is decided at runtime at the start of each base period. A task is released if it has completed its computation and its period $p_{\tau}$ has elapsed since its last release. Note that all tasks are released in the program’s first base period. In Figure 7.5b, task C is allocated $t_C = 100\text{ms} \cdot 0.5 = 50\text{ms}$ and task D is allocated $t_D = 100\text{ms} \cdot 0.5 = 50\text{ms}$ in each base period. It can be verified that tasks C and D can complete their computations in 5 and 2 base periods, respectively, which equates to $2\text{Hz}$ and $5\text{Hz}$. However, preemptive scheduling is required to split the execution of each task and the number of preemptions increases with the number of tasks.
Although tasks are scheduled for their maximum computation time $c_\tau$, their actual computation time may be shorter. Thus, slack can develop at runtime and lead to an under-utilized system. The slack can be used to help mission and non-critical tasks execute at a faster frequency and improve system utilization. This was the insight for the early-release EDF (ER-EDF) approach [155, 157] where each (low critical) task has a set of recurring times specifying when it can be released earlier than usual. Such a task is shown in Figure 7.6 with release time $r_\tau$, maximum period $p_{\tau}^{\text{max}}$, and a set of statically defined early release times $\{k_0, \ldots, k_i\}$. The task completes its computation after $k_1$ and can be considered for early release at $k_2, \ldots, k_i$. The task is only released early if a processor has enough slack to execute the task for its maximum computation time $c_\tau$. Unfortunately, statically defining the early release times that can maximize the use of the slack at runtime is non-trivial. Moreover, the scheduling overheads will increase with the number of early release times that need to be checked. The ER-EDF approach can be applied to the hyper period approach but determining the early release times remains non-trivial. This is because slack can develop at anytime between the statically scheduled task release times. We take a simpler approach to reclaiming slack. Specifically, we use the base period approach to statically schedule all the life and mission critical tasks. At runtime, if a task completes before its allocated time expires, then the next task in the static schedule is executed. Thus, slack will always accumulate at the end of each base period and the slack can be used to execute the non-critical tasks. In the following, we describe the proposed scheduling and some heuristics to further improve system utilization.

### 7.3.3 Static Scheduling Algorithm

A static schedule is obtained by Algorithm 4. Lines 1–3 compute the base period $p_b$ of the tasks. Let us recap the UAV running example described in Figure 7.1. The UAV consists of six tasks with various minimum and maximum frequencies: 4Hz, 10Hz, 20Hz, and 25Hz. Hence, the base period for these tasks is $p_b = \text{GCD}\{\frac{1}{4}, \frac{1}{10}, \frac{1}{20}, \frac{1}{25}\} = \frac{1}{100} = 10ms$. For each life and mission critical task, the portion of computation time that can be allocated in each base period is calculated on lines 5–9. The task’s minimum and maximum utilizations are used to calculate the minimum and maximum portions, $t_\tau^{\text{min}}$ and $t_\tau^{\text{max}}$, respectively. Table 7.3 exemplifies the computation of $t_\tau^{\text{min}}$ and $t_\tau^{\text{max}}$ for the life and mission critical tasks of the UAV example. The function GEN_SCHEDULE (described in Chapter 7.3.4) is then used to find a feasible task-to-processor allocation (line 10). GEN_SCHEDULE returns a static schedule $s_n \in S$ for each processor $n$. Each static
Algorithm 4 Obtains a static schedule for the base period approach.

**Input:** Set of tasks $\Gamma$, and set of processors $N$.

**Output:** Set of static schedules $S$ for all processors.

1: $p_{\text{min}} := \{1/f_{\text{max}}^\tau \mid \tau \in \Gamma, \ z_\tau \neq \text{non-critical}\}$ \quad \triangleright Min periods.
2: $p_{\text{max}} := \{1/f_{\text{min}}^\tau \mid \tau \in \Gamma\}$ \quad \triangleright Max periods.
3: $p_b := \text{GCD}(p_{\text{min}} \cup p_{\text{max}})$ \quad \triangleright Base period.
4: $T := \emptyset$ \quad \triangleright Set of min and max allocation times for each task.
5: for all $\tau \in \Gamma, z_\tau \neq \text{non-critical}$ do \quad \triangleright Life & mission critical tasks.
6: $t_{\text{min}}^\tau := p_b \cdot u_{\text{min}}^\tau$ \quad \triangleright Min time needed during each base period.
7: $t_{\text{max}}^\tau := p_b \cdot u_{\text{max}}^\tau$ \quad \triangleright Max time needed during each base period.
8: $T := T \cup \{t_{\text{min}}^\tau, t_{\text{max}}^\tau\}$
9: end for
10: $S := \text{GENSchedule}(p_b, N, T)$ \quad \triangleright Generate static schedules.
11: return $S$

<table>
<thead>
<tr>
<th>Task $\tau$</th>
<th>Given Parameters</th>
<th>Computed Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$z_\tau$</td>
<td>$c_\tau$</td>
</tr>
<tr>
<td>Navigation</td>
<td>life</td>
<td>75ms</td>
</tr>
<tr>
<td>Stability</td>
<td>life</td>
<td>32.5ms</td>
</tr>
<tr>
<td>VideoStream</td>
<td>mission</td>
<td>28ms</td>
</tr>
<tr>
<td>Avoidance</td>
<td>mission</td>
<td>25ms</td>
</tr>
</tbody>
</table>

Table 7.3: Computed parameters for the life and mission critical tasks of the UAV example. $p_b = 10ms$.

schedule $s_n$ is an ordered list of allocated tasks and execution times. The static scheduling algorithm can be extended to heterogeneous processors by considering each task’s computation time on each heterogeneous processor. This would require $n \in N$ to be an additional parameter of $c_\tau$ and its derived values, such as $u_{\text{min}}^\tau$, $u_{\text{max}}^\tau$, $t_{\text{min}}^\tau$, and $t_{\text{max}}^\tau$.

7.3.4 GenSchedule

We use integer linear programming (ILP) to find a feasible partition of tasks over the processors and a corresponding static schedule. The ILP formulation requires the following inputs: $p_b$, the base period of the tasks; $n \in N$, the set of available processors; and $t_{\text{min}}^\tau, t_{\text{max}}^\tau \in T$, the set of minimum and maximum execution times to allocate. ILP requires all constraints to use integral coefficients. Thus, $p_b$ and all times in $T$ need to be integers. To define the objective function of the ILP formulation, we start by defining the utilization $u_n$ of processor $n$ in every base period. Before formalizing $u_n$, we need to introduce the following terms:

$\alpha$: This denotes the maximum execution time (cost) that a processor may need to spend on preempting a task. This preemption cost arises because a task
The ForeMC Framework

is only executed for a portion of its maximum execution time in each base period. This cost includes the time needed to interrupt the task, save the task’s execution context, and select a different task for execution. WCET analysis can be used to estimate the cost of \( \alpha \).

**\( \beta \):** This denotes the maximum execution time (cost) that a processor may need to spend on managing a task’s delayed communication. For example, using the circular buffer described in Chapter 7.2.2, \( \beta \) would include the cost for a task to send and receive all of its data with the send and release functions. WCET analysis can be used to estimate the cost of \( \beta \). Note that a task’s delayed communication only occurs when it finishes its computation and that tasks can finish in different base periods. Thus, the time needed to resolve delayed communication can vary in each base period. Since all base periods will use the same static schedule, \( \beta \) needs to be a safe upper bound on the time spent resolving delayed communication.

**\( a_{\tau}^{n} \):** A Boolean variable to track whether task \( \tau \) is allocated to processor \( n \), i.e.,

\[
a_{\tau}^{n} = \begin{cases} 
1 & \text{If task } \tau \text{ is allocated to processor } n \\
0 & \text{Otherwise}
\end{cases}
\]

Thus, the utilization \( u_{n} \) of a processor in each base period is the amount of time that it spends executing its allocated tasks and the costs associated with delayed communication and task preemptions. A processor’s utilization cannot exceed the base period:

\[
u_{n} = \sum_{\tau \in \Gamma} \left( a_{\tau}^{n} \cdot (t_{\tau}^{min} + \alpha + \beta) \right)
\]

\[u_{n} \leq p_{b}\]  

Finally, the objective function of the ILP formulation is to maximize the utilization \( U \) of all the processors:

Maximize : 
\[
U = \sum_{n \in N} u_{n}
\]

We constrain the allocation of a task to exactly one processor:

\[
\forall \tau \in \Gamma : \sum_{n \in N} a_{\tau}^{n} = 1
\]

If all processors satisfy equation (7.8), then the tasks are schedulable (Definition 6
7.3 Scheduling ForeMC for Parallel Execution

and equation (7.8)) and a solution to the objective function exists. A static schedule \( s_n \) is constructed for each processor as an ordered list of allocated tasks and execution times. Since the tasks communicate using delayed semantics (Chapter 7.2.2), an arbitrary task order can be used because task dependencies do not exist within each base period. Let \( S \) contain all the static schedules, such that \( \forall n \in N, s_n \in S \). Possible static schedules for the UAV example over two processors are shown in Figure 7.7a. The scheduling of non-critical tasks (e.g., Logging and Sharing) is considered later in Chapter 7.3.5.

The static schedule for processors 1 and 2 in Figure 7.7a contain 0.5\( ms \) and 4.7\( ms \) of slack, respectively. This is because equation (7.7) only uses minimum task execution times \( t_{\tau}^{min} \) to calculate processor utilization. If a processor has slack, i.e., \( u_n < p_b \), then extra execution time can be allocated to the mission critical tasks. This would allow the mission critical tasks to complete their computations earlier and be released more frequently (towards their \( f_{\tau}^{max} \)). The maximum extra execution time that a mission critical task can make use of is bounded by \( x_{\tau}^{max} = (t_{\tau}^{max} - t_{\tau}^{min}) \). In the following, we extend the ILP formulation to utilize the slack on each processor. Let \( x_{\tau}^{n} \) denote the extra execution time that can be allocated
to task $\tau$ on processor $n$:

$$0 \leq x^\tau_n \leq (a^\tau_n \cdot x^\text{max}_\tau) \quad (7.11)$$

The $a^\tau_n$ variable ensures that task $\tau$ only receives extra execution time on its allocated processor, i.e., $x^\tau_n = 0$ when $a^\tau_n = 0$. We update equation (7.7) with the following equation to include the extra execution time:

$$u_n = \sum_{\tau \in \Gamma} \left( a^\tau_n \cdot (t^\min_\tau + \alpha + \beta) + x^\tau_n \right) \quad (7.12)$$

Based on equations (7.11) and (7.12), the slack is allocated arbitrarily and possibly in an *unfair* manner. Figure 7.7b shows an unfair allocation for the UAV example, where the VideoStream task is allocated its entire $x^\text{max}_\tau$ time and the Avoidance task is not. Slack could also be allocated to only a few tasks. Fairness [147] has many definitions and can be measured by various metrics, depending on the application at hand. Thus, fairness can be decided manually by the programmer or automatically by a chosen metric. As an example, we describe a simple fairness constraint based on proportionate fairness [148] for homogeneous processors. The constraint would need to be reformulated for heterogeneous processors. Using $x^\text{max}_\tau$ as the metric, tasks with larger $x^\text{max}_\tau$ are allocated more slack than tasks with smaller $x^\text{max}_\tau$. More precisely, if a task $\tau$ can reclaim more slack relative to another ($x^\text{max}_\tau \geq x^\text{max}_{\tau'}$), then the slack allocated to task $\tau$ on processor $n$ ($x^\tau_n$) is also greater than the slack allocated to the second task on processor $n'$ ($x^\tau_{n'}$), i.e., $x^\tau_n \geq x^\tau_{n'}$. We capture this for any two tasks, $\tau$ and $\tau'$, with the inequality ($x^\text{max}_\tau / x^\text{max}_{\tau'} \leq (x^\tau_n / x^\tau_{n'})$, where $x^\text{max}_\tau \geq x^\text{max}_{\tau'}$, and $n$ and $n'$ are the processors that tasks $\tau$ and $\tau'$, respectively, are allocated to (i.e., $a^\tau_n = 1$ and $a^\tau_{n'} = 1$). Rearranging to remove the divisions, we have the following constraint:

$$\forall n, n' \in N, \forall \tau, \tau' \in \Gamma, x^\text{max}_\tau \geq x^\text{max}_{\tau'}, a^\tau_n = 1, a^\tau_{n'} = 1 :$$

$$x^\tau_{n'} \cdot x^\text{max}_{\tau'} \leq x^\tau_n \cdot x^\text{max}_{\tau} \quad (7.13)$$

Since equation (7.13) is generated for pairs of tasks, it will not scale for large task sets. Thus, this fairness constraint can be omitted for shorter ILP solving time. For the UAV example, possible static schedules with fairly allocated extra execution times are shown in Figure 7.7c.
7.3 Scheduling ForeMC for Parallel Execution

7.3.5 Runtime Scheduling

In every base period, the processors use their static schedules, created by ILP in Chapter 7.3.4, to execute the life and mission critical tasks for up to their allocated execution times. This is called the static scheduling phase. If a task completes its computation without needing all of its allocated time, then the next task in the schedule is selected immediately for execution. After a processor has completed its static schedule, the remaining time in the base period is called slack. This slack is used in a dynamic scheduling phase to execute the non-critical tasks towards their goal frequency and to execute the mission critical tasks further towards their maximum frequency. The static and dynamic scheduling phases that occur on each processor in every base period are shown in Figure 7.8a and can be broken down into three consecutive steps:

**Step 1:** This is the static scheduling phase where the life and mission critical tasks are executed for up to their statically allocated times. Tasks are released at the start of the base period if they have completed their computations and their minimum period $p_{min}$ has elapsed since their last release. The static scheduling phase becomes shorter when tasks complete their computations without needing all of their allocated time. When the static scheduling phase ends, the remaining time in the base period is the slack.

**Step 2:** This is the first half of the dynamic scheduling phase that occurs during the processor’s slack. Non-critical tasks are selected arbitrarily to execute until they complete their computation or until the base period ends. Note that this is the only opportunity for non-critical tasks to execute because
they are not statically scheduled. Non-critical tasks can be executed on any processor. Note that, in some base periods, only a portion of the slack may be needed to complete the computation of all the non-critical tasks.

**Step 3:** This is the second half of the dynamic scheduling phase that occurs when step 2 completes and there is still slack on the processor. Mission critical tasks are selected to continue their execution and improve their execution frequency. Only the mission critical tasks that have completed the static scheduling phase can be chosen for execution in the dynamic scheduling phase. For the dynamic scheduling phase only, each mission critical task can be executed on any processor.

Observe that the duration of each step depends on the actual computation time of the tasks. For example, Figure 7.8b is a trace of a processor’s base period where only mission critical tasks can be executed in the dynamic scheduling phase because all the non-critical tasks have already completed their computations.

### 7.3.6 Heuristics

This section describes some heuristics for improving the use of slack in the dynamic scheduling phase. For the second scheduling step, to ensure all non-critical tasks get equal opportunities to execute, the non-critical tasks that have received the least amount of cumulative slack are executed first. For the third scheduling step, to maintain proportionate progress [148] among the mission critical tasks, the mission critical tasks with the least improvement in execution frequency (compared to their maximum possible improvement) are executed first. A task’s improvement in execution frequency is measured by:

\[
    f_{\tau}^{\text{improve}} = \frac{f_{\tau}^{\text{avg}} - f_{\tau}^{\text{min}}}{f_{\tau}^{\text{max}} - f_{\tau}^{\text{min}}}
\]

(7.14)

where, \(f_{\tau}^{\text{avg}}\) is task \(\tau\)’s average execution frequency at runtime. Thus, tasks with lower \(f_{\tau}^{\text{improve}}\) values are executed first.

The slack time can be used more effectively when the mission critical tasks are scheduled before the life critical tasks in the static scheduling phase. We explain this using Figure 7.9. In Figure 7.9a, processor 1 statically executes a life critical task \(A\) and a mission critical task \(B\) for the entire base period. Processor 2 statically executes a life critical task \(C\), which uses only a fraction of its allocated time. Assuming that the system does not have any non-critical tasks, the slack on processor 2 cannot be used. By contrast, in Figure 7.9b, processor 1 statically
executes the mission critical task \( B \) before the life critical task \( A \). After task \( B \) has executed for its allocated time, it can continue its execution in processor 2's dynamic execution phase. Hence, the slack time is used more effectively.

When there are no non-critical or mission critical tasks for a processor to execute in the dynamic scheduling phase, the processor’s slack will be forfeited. Instead of forfeiting the slack, it is used to execute the life critical tasks. Although the release frequencies of life critical tasks cannot be improved, they will complete their computations earlier and produce slack in a future base period. In the future base period, hopefully a non-critical or mission critical task can use the slack.

## 7.4 Discussion

We proposed an approach for the design of mixed-criticality systems by relaxing the well known synchronous approach. The proposed approach allows the modeling of fixed-rate tasks (similar to the synchronous model) and variable-rate tasks. The proposed approach allows three levels of criticalities to be modeled: life critical (tasks with constant frequencies), mission critical (tasks supporting bounded variability of their frequencies), and non-critical (tasks with goal frequencies, which may or may not be satisfied). To our knowledge, the proposed approach is a novel
synchronous extension for supporting mixed-criticality task sets. It is unlike the earlier task model proposed by Baruah [154, 169] which supports high and low critical tasks with fixed frequencies. Although the high and low critical tasks are similar to our life and non-critical tasks, Baruah’s model does not support tasks with variable frequencies like our mission critical tasks. In addition, the scheduling approach of Baruah does not try to maximize the runtime utilization of the processors and fairness is not considered in the execution of low critical tasks. The other major contributions of the proposed approach are:

- We proposed a static (offline) scheduling algorithm that partitions the tasks over the processors such that all the life and mission critical tasks can complete their computations within their maximum periods. The mission critical tasks are given extra execution time whenever possible to allow them to execute faster than their minimum frequency and to maximize the processor utilization. The proposed formulation is done using ILP with fairness constraints to control the distribution of slack to the mission critical tasks.

- We proposed a complementary dynamic (online) scheduling algorithm to reclaim the buildup of slack in each base period because the static scheduling only considers worst-case computation times. The non-critical tasks are dynamically scheduled to give them the opportunity to reach their goal frequencies. If there are no non-critical tasks to schedule, then the mission critical tasks are dynamically scheduled to allow them to execute toward their maximum frequencies. Several runtime heuristics were proposed to ensure fairness in task executions and to maximize the recovery of slack.

- We evaluated the proposed approach relative to the ER-EDF [157] scheduling approach for mixed-criticality systems in Chapter 8. Unlike ER-EDF, the proposed approach does not need early release times to be specified by the user and achieves superior runtime utilization, task frequency improvements, and fairness. These benefits are elaborated further in Chapter 8.

It is worth discussing the other real-time scheduling approaches developed for mixed-criticality task sets. Compared to ZS-QRAM [163], our proposed scheduling approach maximizes system utilization in a complementary manner by increasing a task’s resource usage beyond its static allocation when resources become available, rather than reducing a task’s resources usage from its initial allocation when resources become scarce. The use of marginal utility in ZS-QRAM can be used as a scheduling heuristic in our approach. The TDMA work by Pop et al. [274] integrates the scheduling of mixed time-triggered and event-triggered task sets
with separate dynamic and static slots for communication. Steiner [275] integrates time-triggered network traffic with non-time-triggered traffic. Although the work by both Pop et al. and Steiner address a different problem (scheduling over communication networks), there are similarities in that they statically allocate time-triggered traffic. A major difference is that all our tasks are implicitly time-triggered (none are event-triggered) and that the periods of the time-triggered tasks can be assigned bounds. Our current approach does not yet consider distributed embedded systems with a shared communication network. We consider it as future work to integrate our approach with reliable network protocols.

Our proposed ForeMC framework uses buffers with statically bounded buffer sizes for lossless communication. Goddard and Jeffay [276, 277, 278] describe the management of buffer sizes in multi-rate programs defined as Processing Graphs mapped to the Rate-Based Execution [151] task model. Processing Graphs are similar to SDF [121] and task execution rates depend on the (statically defined) task communication rates. Buffer sizes are bounded by the amount of data that can be produced before they can be consumed. Since data is made available as soon as it is produced, the bound also depends on the task scheduling order. In contrast, our proposed lossless buffering decouples the task execution frequencies from their communication frequencies, allowing (mission critical) tasks to vary their execution frequency depending on the processor utilization. By using delayed semantics, the bounding of each buffer is unaffected by task scheduling order.

The ForeMC framework can be useful for the FlexPRET [161] processor when the number of tasks is more than the number of hardware threads. The proposed scheduling approach can be a useful component in a PRET infrastructure [279].
This chapter quantitatively assesses the following objectives of the proposed scheduling approach of the ForeMC framework: task schedulability, processor utilization, task execution frequency, and scheduler fairness. A simulation-based approach is used to compare the performance characteristics of the proposed approach (Chapter 7.3) with those of the early-release earliest deadline first (ER-EDF) approach [157]. The scheduling behavior of both approaches are explored by varying the proportions of life, mission, and non-critical tasks during simulation. Additional results are provided for the scalability of the ILP-based static scheduling algorithm (Chapter 7.3.3), and the expected task preemption costs.

The ER-EDF task model extends the earliest deadline first (EDF) algorithm to support high and low critical tasks. The tasks are statically partitioned over the processors. High critical tasks are released at constant periods and must complete their computations before their next release times (implicit deadlines). Thus, high critical tasks can be mapped directly to our proposed life critical tasks. A low critical task, shown in Figure 8.1a, has a maximum release period \( p_{\tau}^{\text{max}} \) and a set of early release times \( \{k_0, \ldots, k_i\} \). If a low critical task completes its computation before \( p_{\tau}^{\text{max}} \), then it can be considered for release at one of its next early release times. Using global scheduling, the task is released early if a processor has enough slack to execute the task for its maximum computation time \( c_{\tau} \). A low critical task can be mapped to our proposed mission critical tasks in the following manner. Figure 8.1b shows a mission critical task with its minimum and maximum periods, \( p_{\tau}^{\text{min}} \) and \( p_{\tau}^{\text{max}} \), respectively. By construction (Chapter 7.3.3), \( p_{\tau}^{\text{min}} \) and \( p_{\tau}^{\text{max}} \) coincide with the start of a base period. Moreover, the time between \( p_{\tau}^{\text{min}} \) and \( p_{\tau}^{\text{max}} \) is an integral multiple of the base period \( p_b \). Hence, the early release times of a mission critical task are at the start of every base period between \( p_{\tau}^{\text{min}} \) and \( p_{\tau}^{\text{max}} \). Thus, a low critical ER-EDF task can be mapped to our proposed mission critical task when 1) they have the same maximum period \( p_{\tau}^{\text{max}} \), 2) the first early release
time of the low critical task is equal to $p^c_{\text{min}}$ of the mission critical task, and 3) the subsequent early release times of the low critical task occur every $p_b$. We refer to the high and low critical tasks of ER-EDF as simply life and mission critical tasks. Note that ER-EDF does not support our notion of non-critical tasks. The ER-EDF results [157] show best schedulability when the tasks were partitioned over the processors using the first-fit decreasing-criticality heuristic. We refer to this as the ER-EDF-FF approach. According to this, the life critical tasks are allocated before the mission critical tasks using first-fit (FF). Tasks of the same criticality are sorted by minimum task utilization $u^c_{\text{min}}$ (equation (7.2)) in descending order. We also compare against the traditional EDF approach.

We follow the simulation-based evaluation approach of ER-EDF by Su et al. [157] and describe how we generate our task parameters. A task’s maximum utilization $u^c_{\text{max}}$ is generated between 0.05 and 0.5 (5% and 50%) in a uniform distribution. For the evaluation, the system’s base frequency $f_b = 1/p_b$ is limited to between 100Hz and 1000Hz. Thus, $f_b$ is generated between 100Hz and 1000Hz in a uniform distribution and a task’s $f^c_{\text{min}}$ and $f^c_{\text{max}}$ are random divisors of $f_b$. The generated $u^c_{\text{max}}$, $f^c_{\text{min}}$, and $f^c_{\text{max}}$ values are inputs to our static scheduling algorithm (Chapter 7.3.3). For ER-EDF, the early release times of the mission critical tasks are generated to be equivalent to those of our proposed approach, e.g., Figure 8.1b. Only life and mission critical tasks are generated for each task set and the proportion of life critical tasks is denoted by $\text{prop}(\text{life})$. Each task set $\Gamma$ is generated according to a normalized system utilization [157] $U^\text{max} = \frac{U^\text{max}}{N}$, where

$$U^\text{max} = \text{Max}\{ U^\text{c}(\text{life}), U^*_{\Gamma}(\text{life}) + U^\text{min}(\text{mission}) \}$$

and $N$ is the number of processors in the system. $U^\text{max}$ tries to approximate the
average runtime utilization of the task set. It is the maximum of either $U_T(\text{life})$, the total utilization of the life critical tasks, or $U_T^*(\text{life}) + U_T^{\text{min}}(\text{mission})$, the total reduced utilization of the life critical tasks plus the total minimum utilization of the mission critical tasks. $U_T(\text{life})$ and $U_T^{\text{min}}(\text{mission})$ were defined by equations (7.4) and (7.5) in Chapter 7.3, respectively:

$$U_T(\text{life}) = \sum_{\tau \in \Gamma, \ z_r = \text{life}} (u_{T_r}^{\text{min}})$$

$$U_T^{\text{min}}(\text{mission}) = \sum_{\tau \in \Gamma, \ z_r = \text{mission}} (u_{T_r}^{\text{min}})$$

For simulation purposes only, $U_T^*(\text{life})$ is the total utilization of the life critical tasks at reduced computation times and, therefore, reduced utilization $u_r^*$:

$$U_T^*(\text{life}) = \sum_{\tau \in \Gamma, z_r = \text{life}} u_r^*$$ (8.2)

where, $u_r^*$ is generated between $u_r^{\text{min}}/8$ and $u_r^{\text{min}}$ in a uniform distribution. We do not consider the effects of scheduling overheads in our comparisons. The proposed scheduling approach uses the heuristics detailed in Chapter 7.3.6.

8.1 Schedulability

We evaluate task schedulability for the proposed and ER-EDF-FF approaches. The performance metric we use is the acceptance ratio [157], defined as the proportion of generated task sets that are schedulable. We use Gurobi [280] (version 5.6) to solve the ILP constraints of the proposed static scheduling approach. On average, 118.9 constraints are generated per task set and Gurobi takes more than a day to find solutions for some task sets. If the fairness constraint (equation (7.13)) is omitted, then an average of 66.5 constraints are generated per task set, requiring on average less than a minute to solve. As a compromise, we generate ILP constraints with fairness and allow Gurobi one minute to find a (possibly locally optimal) solution. If a solution cannot be found after one minute, we consider the task set unschedulable. Figures 8.2a–8.2c show the acceptance ratio under varying normalized system utilization and $prop(\text{life})$. 10,000 task sets were generated for each data point and attempted to be scheduled on a system with four (homogenous) processors. The acceptance ratio decreases with increasing normalized system utilization because equation (8.1), used to generate the task sets, is more optimistic than the test for task schedulability (Definition 6). Hence, as the
Figure 8.2: Acceptance ratio when the normalized system utilization and \( \text{prop}(\text{life}) \) are varied.
normalized system utilization increases, more unschedulable task sets are generated. When \( \text{prop}(\text{life}) = 50\% \), the greatest proportion of unschedulable task sets is generated, causing both approaches to reject the most task sets. As expected, the first-fit heuristic used by ER-EDF-FF rejects more task sets than the proposed approach. This is because the proposed approach uses ILP to find better task-to-processor allocations. Figure 8.2d highlights this by showing the result of subtracting the acceptance ratio of ER-EDF-FF from the proposed approach. The proposed approach can accept up to 15% more task sets than ER-EDF-FF.

### 8.2 Effects of Varying \( \text{prop}(\text{life}) \)

We evaluate the effects of varying \( \text{prop}(\text{life}) \) on the system runtime utilization and the execution frequency of mission critical tasks. We execute each task set for a minimum of 1,000 base periods. Each time a task is released, its actual computation time is chosen uniformly between \( 0.8 \cdot c_r \) and \( c_r \). The normalized system utilization is set to 50% because schedulable task sets are harder to generate at higher utilization. For a fairer comparison, we allow ER-EDF to use the task-to-processor allocations found by the proposed approach (Chapter 7.3.4) and call this the ER-EDF-ILP approach.

Figures 8.3a and 8.3b show the system runtime utilization when \( \text{prop}(\text{life}) \) is varied for 4 and 8 (homogeneous) processors. 100 task sets were generated for each data point. System runtime utilization is the average proportion of time that the system spends executing tasks. The system runtime utilization that is achieved without early releases is shown by the EDF result. The much higher system runtime utilization of the proposed and ER-EDF approaches show the benefit of releasing tasks early. The average (geometric mean) system runtime utilization of the proposed approach is 5.38% better than ER-EDF-ILP. For the proposed approach, slack always accumulates at the end of each base period and is easier for tasks to access than in ER-EDF. The system runtime utilization of all the approaches peak when \( 50\% \leq \text{prop}(\text{life}) \leq 60\% \). The system runtime utilization converges to the normalized system utilization of 50% when \( \text{prop}(\text{life}) \) increases to 100% because less mission critical tasks are available to use the slack.

Figures 8.3c and 8.3d show the overall improvement in execution frequency for all mission critical tasks when \( \text{prop}(\text{life}) \) is varied for 4 and 8 processors. The overall improvement is defined as:

\[
\text{f}_{\text{improve}}^{\text{mission}} = \frac{\sum_{\tau \in \Gamma, \zeta_r = \text{mission}} (f_{\tau}^{\text{avg}} - f_{\tau}^{\text{min}})}{\sum_{\tau \in \Gamma, \zeta_r = \text{mission}} (f_{\tau}^{\text{max}} - f_{\tau}^{\text{min}})}
\]  

(8.3)
Figure 8.3: System runtime utilization, overall frequency improvement, and fairness when \( \text{prop}(\text{life}) \) is varied. Normalized system utilization = 50%.
8.2 Effects of Varying \(\text{prop} (\text{life})\)

where, \(f^\text{avg}_{\tau}\) is task \(\tau\)'s average execution frequency over its entire execution history. A higher value of \(f^\text{improve}_{\text{mission}}\) means a better overall improvement. The results show that \(f^\text{improve}_{\text{mission}}\) increases until \(\text{prop} (\text{life}) = 100\%\), when no mission critical tasks are generated. The value of \(f^\text{improve}_{\text{mission}}\) decreases slightly around \(50\% \leq \text{prop} (\text{life}) \leq 60\%\) because the system runtime utilization, without releasing tasks early, is already near its peak. This was illustrated by the EDF result in Figures 8.3a and 8.3b. The value of \(f^\text{improve}_{\text{mission}}\) increases greatly when \(\text{prop} (\text{life})\) increases towards \(100\%\). This is because Figures 8.3a and 8.3b showed that the slack increases when \(\text{prop} (\text{life})\) increases to \(100\%\). The slack is also shared among fewer mission critical tasks. The average (geometric mean) improvement in execution frequency for the proposed approach is \(5.91\%\) better than ER-EDF-ILP.

Figures 8.3e and 8.3f show how fairly the execution frequencies of the mission critical tasks are improved by when \(\text{prop} (\text{life})\) is varied for 4 and 8 processors. We define fairness as how well all the mission critical tasks can improve their execution frequency by the same proportion:

\[
\text{fairness} = \frac{\sum_{\tau \in \Gamma_{\zeta=\text{mission}}} |f^\text{avg,improve}_{\tau} - f^\text{improve}_{\tau}|}{\text{Number of mission critical tasks}}
\]  

(8.4)

where, \(f^\text{improve}_{\tau}\) was already defined by equation (7.14) in Chapter 7.3.6:

\[
f^\text{improve}_{\tau} = \frac{f^\text{avg}_{\tau} - f^\text{min}_{\tau}}{f^\text{max}_{\tau} - f^\text{min}_{\tau}}
\]

and

\[
f^\text{avg,improve}_{\tau} = \frac{\sum_{\tau \in \Gamma_{\zeta=\text{mission}}} (f^\text{improve}_{\tau})}{\text{Number of mission critical tasks}}
\]  

(8.5)

is the average overall frequency improvement of all the mission critical tasks. A completely fair improvement in execution frequencies results in \(\text{fairness} = 0\%\) because all the \(f^\text{improve}_{\tau}\) values are identical. A completely unfair improvement results in \(\text{fairness} = 50\%\) because all the \(f^\text{improve}_{\tau}\) values are spread apart with \(f^\text{avg,improve}_{\tau}\) in the middle. As expected the proposed approach is fairer than the other approaches when \(0\% \leq \text{prop} (\text{life}) \leq 60\%\). Fairness can be improved with better fairness constraints and heuristics. All approaches become unfair when \(60\% \leq \text{prop} (\text{life}) \leq 90\%\) because some task sets have less mission critical tasks than the number of processors and the fairness heuristics (Chapter 7.3.6) are less effective on such task sets. All approaches are very fair when \(\text{prop} (\text{life})\) is around \(100\%\) because less than two mission critical tasks are generated in each task set.
8.3 Effects of Varying Normalized System Utilization

We repeat the evaluation described in Chapter 8.2, except we vary the normalized system utilization and set \( prop(life) = 50\% \). We vary the normalized system utilization from 0% to 50% because schedulable task sets are harder to generate at higher utilization.

Figure 8.4a shows system runtime utilization under varying normalized system utilization. As expected, the system runtime utilization is always higher than the normalized utilization. The proposed approach achieves higher system runtime utilization than the other approaches when the normalized system utilization increases. The average (geometric mean) system runtime utilization of the proposed approach is 3.85% higher than ER-EDF-ILP and 6.63% higher than ER-EDF-FF.

Figure 8.4b shows the overall improvement in execution frequency for all mission critical tasks \( (f^\text{improve}_\text{mission} \text{ defined by equation (8.3)}) \) under varying normalized system utilization. Note that the available amount of slack in the system decreases when the normalized system utilization is increased. Hence, less slack will be available for mission critical tasks to improve their execution frequency. This is reflected in Figure 8.4b by the decreasing values of \( f^\text{improve}_\text{mission} \). The average (geometric mean) \( f^\text{improve}_\text{mission} \) of the proposed approach is 4.87% higher than ER-EDF-ILP and 11.85% higher than ER-EDF-FF. The proposed approach achieves better \( f^\text{improve}_\text{mission} \) than the other approaches when the normalized system utilization increases.

Figure 8.4c shows how fairly \( (\text{fairness} \text{ defined by equation (8.4)}) \) the execution frequencies of mission critical tasks are improved as normalized system utilization is increased. For all approaches, the improvement is most unfair at around 30% normalized system utilization. This may be because low normalized system utilization allows all mission critical tasks to execute near their maximum frequency. Between 10% and 50% normalized system utilization, some mission critical tasks may be released more frequently if they are the only available task. After 50% normalized system utilization, the diminishing slack may limit how quickly mission critical tasks can complete. The average (geometric mean) fairness of the proposed approach is, 2.55% better than ER-EDF-ILP and 5.33% better than ER-EDF-FF.

8.4 Effects of Including Non-Critical Tasks

We repeat the evaluation described in Chapter 8.2, except we introduce non-critical tasks into the generated task sets. Let \( prop(mission) \) and \( prop(non\text{-critical}) \) de-
8.4 Effects of Including Non-Critical Tasks

Figure 8.4: System runtime utilization, overall frequency improvement, and fairness when normalized system utilization is varied. \( prop(life) = 50\% \).

Figure 8.5: System runtime utilization, overall frequency improvement, and fairness when \( prop(non\text{-}critical) \) is varied. Norm. sys. utilization = 50%.
note the proportion of mission and non-critical tasks generated in each task set. For the evaluation, we vary $prop(\text{non-critical})$ from 10% to 90% and leave equation (8.1) unchanged. Equal proportions of life and mission critical tasks form the remaining tasks, i.e., $prop(\text{life}) = prop(\text{mission})$. The normalized system utilization is set to 50%. Thus, when $prop(\text{non-critical})$ increases from 10% to 90%, a relatively constant number of life and mission critical tasks but an increasing number of non-critical tasks are generated.

Figure 8.5a shows that the system runtime utilization reaches 100% quickly with increasing $prop(\text{non-critical})$. This is because more non-critical tasks are generated to use the slack. Figures 8.5b and 8.5c show the values of $f_{\text{improve}}$ and fairness for the mission critical tasks. Both $f_{\text{improve}}$ and fairness are relatively constant as $prop(\text{non-critical})$ increases. This is because of two factors. First, mission critical tasks are rarely scheduled in the dynamic scheduling phase because non-critical tasks have higher priority (Chapter 7.3.6). Second, the static scheduling algorithm (Chapter 7.3.4) allocates slack to the mission critical tasks in a proportionately fair manner. Hence, the $f_{\text{improve}}$ and fairness values shown in Figures 8.5b and 8.5c are mostly due to the static schedule. Note that when $prop(\text{non-critical})$ is 10%, the system runtime utilization is less than 100%. This means all the mission and non-critical tasks get scheduled during the dynamic scheduling phase. As a result, $f_{\text{improve}}$ is at the highest (19.8%) when $prop(\text{non-critical})$ is 10%.

8.5 ILP Scalability

Gurobi solves ILP problems iteratively by finding better locally optimal solutions until the globally optimal solution is found. It is possible to run Gurobi until it finds the first locally optimal solution for a substantially shorter solving time. Running Gurobi in this mode is similar to using a heuristic because the first locally optimal solution can be treated as an approximation to the final globally optimal solution. To demonstrate this approach, we used Gurobi to solve the ILP constraints of Chapter 7.3.4 (including fairness) on 250 randomly generated task sets, each containing 2 to 50 tasks. Figure 8.6 shows that Gurobi finds locally optimal solutions for all task sets in under one second, but an increasing amount of time is needed to find the globally optimal solutions.
8.6 Response Times and Preemptions

The use of delayed communication in our approach means that the outputs computed by a task are only made available at the end of the task’s period. Thus, with respect to the timing of outputs, a task’s response time correlates to its execution frequency. For a life critical task, its response time is constant because its minimum and maximum frequencies, $f_{min}$ and $f_{max}$, are equal. For a mission critical task, its execution frequency is statically guaranteed to meet $f_{min}$ but can improve towards $f_{max}$ at runtime. Hence, the response time is bounded by $f_{min}$ and $f_{max}$. Figures 8.3c and 8.3d demonstrate the ability of mission critical tasks to improve their execution frequency. For a non-critical task, it only tries to meet its response time because it only has a goal frequency.

The cost of preempting tasks cannot be ignored when implementing a system. Thus, to gauge these costs, we repeat the evaluation described in Chapter 8.2, but vary the normalized system utilization with $prop(life) = 50\%$ and record the average number of preemptions that occur on each processor. A preemption is recorded whenever a task is interrupted to allow other tasks or the scheduler to execute. Figure 8.7 shows that the proposed approach requires nearly twice the number preemptions than ER-EDF. Thus, in return for achieving higher system utilization and frequency improvements, the proposed approach may incur higher preemption penalties than ER-EDF. However, the preemption cost in terms of execution time depends on how efficiently both approaches are implemented.
8.7 Discussion

This chapter has presented benchmark results that evaluated the task schedulability, processor utilization, task execution frequency, and scheduler fairness of the ForeMC framework. Results showed that the proposed static scheduling algorithm accepts a greater range of task sets than ER-EDF-FF. This is because ILP is used to find better task-to-processor allocations than the first-fit heuristic used by ER-EDF-FF. On average, our proposed scheduling approach was 3.03% fairer than ER-EDF while achieving 5.38% higher system utilization and 5.91% higher execution frequencies. Unlike ER-EDF, our proposed scheduling approach does not need early release times to be specified by the user and achieves superior runtime utilization, task frequency improvements, and fairness. Our task model supports more levels of task criticalities than ER-EDF: life, mission and non-critical.

Due to the nature of base period scheduling, the number of task preemptions required by the proposed approach appears to be much higher than ER-EDF. The impact of higher preemption costs on the processor utilization requires further investigation on a real implementation. Overall, the benchmarking demonstrates great promise in ForeMC’s relaxation of the synchrony hypothesis for the development of mixed-criticality CPSs.
Cyber-physical systems are dominating the world of electronics and are assimilating themselves into our everyday lives. Wearable devices, such as smart glasses and watches, can already monitor and enhance the wearer’s well being in real-time. A rich ecosystem of services will emerge when the devices network and share biometric data with each other. This will lead to a diverse range of mixed-criticality CPSs becoming available with many being certified against stringent safety standards. Today, programmers can choose from a range of languages, tools, and hardware platforms to develop intricate CPSs. With each passing day, the devices become more and more complex and their development can only be sustained in the near term by current methodologies and technologies. The general practice is to program an embedded multi-core ARM processor with C and Pthreads and use an RTOS to manage the execution. We envision a future of programming massively parallel time-predictable processors with a C-like, deterministic, parallel language. The execution could be compiled for bare-metal or managed by a time-predictable RTOS. To bridge the gap between what is achievable today and in the future, considerable advances are needed in languages, tools, and hardware platforms. Chapter 2 of this thesis reviewed three key facets of developing future CPSs: parallel programming, predictable parallel hardware architectures, and static timing analysis. This thesis makes inroads to the future development of CPSs and attempts to unify the various research fields by proposing:

The ForeC language for deterministic, parallel, and reactive programming of parallel architectures. Chapter 3 provided an in-depth description of ForeC and, unlike existing C-based synchronous languages, it is designed specifically for parallel programming. The semantics of ForeC is designed to give programmers the ability to express many forms of parallel patterns while ensuring that ForeC programs can be compiled efficiently for parallel execution and be amenable to static timing analysis. This required the semantics,
compiler, and analyzer to be designed in tandem. ForeC’s main innovation revolves around its shared variable semantics that provides thread isolation and deterministic communication. The behavior of a shared variable can be tailored to the application at hand by specifying a suitable *combine function* and *policy*. All ForeC programs are correct by construction and deadlock-free because mutual exclusion constructs are not needed. The formal semantics greatly simplifies the understanding and debugging of parallel programs.

Chapter 4 presented a compilation approach that used non-preemptive static thread scheduling. The key strategy was to preserve the ForeC threads and use light-weight context-switching and simple scheduling routines to preserve the ForeC semantics. Modular code is intentionally generated to ensure scalable static timing analysis using reachability.

The **ForeMC framework** for defining mixed-criticality tasks and associated task scheduling. In anticipation of the growing abundance of CPSs in everyday life, the ForeMC framework allows the synchronous programming community to venture into the realm of mixed-criticality systems. Chapter 7 introduced two new levels of task criticality (mission and non-critical) to the synchronous model of computation. This was achieved by relaxing the synchronous task model to support tasks with bounded (mission critical) or unbounded (non-critical) tolerances to deadline misses. A lossless communication model was proposed to allow life and mission critical tasks to communicate deterministically at variable frequencies. As size, weight, and power (SWaP) concerns are becoming more important for end-users, the proposed hybrid (static and dynamic) task scheduling approach enables high system utilization and fairness among the tasks.

The **ForeCast analyzer** for verifying the timing properties of parallel programs. Chapter 5 revealed the novel use of the reachability technique to compute the WCET of all the reachable ticks. While traversing the program’s CCFG, ForeCast simultaneously resolves the execution times of each core due to instruction execution, shared bus delays, synchronization delays, and thread scheduling behavior. We described the ability to trade-off WCRT precision with analysis time by using fine-grained or coarse-grained reachability. By using sensible methods to reduce the program’s CCFG, the state-space explosion problem could be alleviated greatly for fine-grained reachability. This allows for scalable and precise WCRT computation.

This thesis makes several significant contributions. ForeC is the first synchronous
language designed specifically for parallel programming. ForeMC is the first framework that relaxes the synchrony hypothesis to allow the design of mixed-criticality systems. ForeCast is the first static timing analyzer for parallel synchronous programs that considers the effects of software-based synchronization and thread scheduling. Together, ForeC, ForeMC, and ForeCast form an integrated tool chain that helps to simplify the CPS development process.

9.1 Future Directions

Although ForeC, ForeMC, and ForeCast make significant contributions for CPS design, many aspects can be further improved and developed. Key aspects are discussed in this section.

9.1.1 Formal Semantics

The ForeMC framework paves the way for programming mixed-criticality systems with synchronous languages. The appeal of synchronous languages is their sound mathematical framework and suitability to formal verification. Thus, formal semantics of ForeMC needs to be developed to ensure wide acceptance within the synchronous language community. ForeMC prescribes a different notion of task synchronization and communication. Thus, the ForeC semantics can be used to describe the behavior of each task. An SDF-like semantics [121] could be used to describe the behavior of task synchronization and communication, except a task consumes its entire buffer when it is released. Asynchronous semantics could be used to describe the release of mission and non-critical tasks because their releases depend on the system’s runtime utilization.

9.1.2 Thread Scheduling

The ForeC compiler can be improved to generate more efficient code that remains amenable to static timing analysis. In particular, different static scheduling strategies could be explored for different design patterns. Currently, scheduling priorities are assigned to ForeC threads by traversing the thread hierarchy in a depth-first manner. However, assigning scheduling priorities in a breadth-first manner could produce more efficient schedules in some cases.

For better average-case performance, e.g., for desktop execution, the ForeC compiler can be improved to generate dynamically scheduled code for desktop execution. Although benchmarking showed that the statically scheduled ForeC
programs executed competitively compared to OpenMP, the ForeC implementa-
tion uses busy waiting to implement blocking reads in the scheduling routines. On
a desktop, where multiple programs may be running, a core is unable to execute
other programs while it is busy waiting for a blocking read to succeed. Thus, busy
waiting is an inefficient use of a core’s execution time. Blocking reads can be re-
placed with waits and signals on condition variables. The thread pooling approach
of the existing ForeC compiler can be extended to dynamically schedule the ForeC
threads to load balance the workload on the available cores.

The ForeC compiler can also be extended to support the ForeMC framework.
The implementation of ForeMC’s hybrid scheduling approach has already begun
on the time-predictable PTARM processor described in Chapter 1.4.2. Recall that
PTARM supports the execution of four independent hardware threads. PTARM
simplifies the programming of time-dependent behaviors, such as the static task
scheduling, by including the following timing instructions in its ISA. Note that
all the hardware threads in PTARM can independently use the following timing
instructions. The \texttt{get\_time} instruction retrieves the current time of the processor.
The \texttt{exception\_on\_expire} instruction takes a timestamp as input and will inter-
rupt the hardware thread when the processor’s time exceeds the timestamp. The
interference can be disabled by using the \texttt{deactivate\_exception} instruction. The
\texttt{delay\_until} instruction takes a timestamp as input and will block the hardware
thread until the processor’s time exceeds the timestamp. These timing instruc-
tions use nanoseconds as the unit of time, which allows precise timing behaviors to
be specified. Thus, these timing instructions allow us to implement a light-weight
task scheduler on each hardware thread. The statically allocated execution time
for each task can be enforced by the \texttt{exception\_on\_expire} instruction. The base
period can be enforced by the \texttt{exception\_on\_expire} instruction when tasks are
dynamically scheduled during the slack, or by the \texttt{delay\_until} instruction when
no tasks can be dynamically scheduled. The resulting implementation will allow
us to quantify the cost of task preemptions and scheduling overheads of ForeMC’s
hybrid scheduling approach.

Other scheduling approaches for ForeMC can be explored. The static schedule
of each base period could be modified to allow task migration across the processors
to improve task release frequencies and system utilization. For this approach, a
sequence of \textit{b}-number of static schedules could be constructed for a correspond-
ing sequence of \textit{b}-number of base periods. Tasks could be allocated to different
processors in each of the static schedules. At runtime, the sequence of schedules
is repeated after every \textit{b}-number of base periods. The fairness of task execution
frequencies could be improved by exploring other notions of fairness and heuristics. The allocation of slack among tasks could be left as a completely runtime objective to see if better fairness could be achieved. The static schedules would simply be constructed with only the minimum task utilizations.

As benchmarking has identified, ForeMC’s proposed hybrid scheduling approach incurs nearly twice the number of preemptions as that of ER-EDF. Work has begun on developing an alternate non-preemptive dynamic scheduling approach based on interval scheduling \[281\]. In interval scheduling, the execution of a task is called an interval and is defined by a start time and an execution time. The scheduler is given the intervals of all the tasks and must find a schedule for the minimum number of processors. Task intervals on the same processor must not overlap and the execution of an interval cannot be preempted. Harder interval scheduling problems consider the following runtime behaviors: intervals changing their start and execution times, or inserting new intervals into the system. The recent work of Gavruskin et al. \[282\] showed that, by organizing the idle time between the intervals into a tree structure, the optimal schedule can be updated at runtime in \(O(d + \log n)\) worst-case time, where \(d\) and \(n\) relate to the intervals being scheduled. An interval scheduling approach for ForeMC could have substantially less overheads than that of the current hybrid approach. However, a reduction in system utilization is expected because the interval scheduling is non-preemptive.

There are also great opportunities for developing an energy-aware thread scheduling algorithm for both ForeC and ForeMC. We envision an algorithm that trades off execution time with energy consumption and quality of service.

### 9.1.3 Static Analysis

ForeCast’s pruning of infeasible paths can be improved by tracking the values of private and shared variables. This can follow the approach of Kuo et al. \[252\], which integrates variable tracking into their reachability analysis of PRET-C synchronous programs on single-cores. That approach would have to be modified to support ForeC’s shared variable semantics. Specifically, the resynchronized shared variables and their possible values at the end of the tick would have to be modeled. ForeCast has parameterized models of the scheduling routines and it should be straightforward to expand the models to track the resynchronized values.

ForeCast’s timing model of the hardware architecture can be expanded to consider more complicated and speculative features of general-purpose processors. Examples include caches, speculative pipelines, high-performance buses, and dynamically allocated scratchpads. This would help broaden the use of ForeC and
ForeCast to the programmers wishing to work with speculative processors.

ForeCast’s timing results can be integrated with the ForeC compiler to automatically refine a program’s thread-to-core allocation and minimize its WCRT. Allocations were manually refined in our benchmarking by reviewing the worst-case execution trace reported by ForeCast. The strategy was to avoid long blocking times by distributing child threads among the cores and to balance the workloads of each core. As a first step, ForeCast could be improved to suggest better thread-to-core allocations to the user. The next step would be to feed the suggested allocations into the ForeC compiler to generate a more efficient static schedule. This process could be repeated until a shorter WCRT cannot be obtained. Eventually, we hope to have a WCRT-aware ForeC compiler.

ForeCast can be extended to analyze the worst-case energy consumption (WCEC) of ForeC and ForeMC programs. Many upcoming wearable devices will have limited energy capacities and energy use could become a correctness criterion for devices with safety-critical concerns. It is foreseeable that WCRT and worst-case energy consumption (WCEC) analysis would be used to optimize a program’s space, time, and energy usage.

9.1.4 Integrated Development Environment for CPS Development

An integrated development environment (IDE) is an application that provides a comprehensive set of features for maximizing programmer productivity. An IDE that unifies ForeC, ForeMC, and ForeCast together with the following features would be conducive to the future development of CPSs.

A ForeC and ForeMC code editor that supports the usual editing facilities such as syntax highlighting, code completion, management of project files, code debugger, and one-click compilation of programs. An option for programmers to visualize their code as an equivalent CCFG would help them comprehend the parallelism in their code. The visualizer could also display the current thread-to-core allocations and allow the programmer to graphically change the allocations. Each code statement could be annotated with their execution time to give the programmer a sense of time.

A ForeCast reporter that visualizes the static timing results of the ForeCast analyzer. The computed WCRT could be visualized as an execution trace showing the execution times of the threads and blocking times of the scheduling routines. By maintaining traceability between the source code and com-
9.2 Final Remarks

Piled code, the WCRT could be highlighted in the source code or on an equivalent CCFG representation of the code. Suggestions could be provided on how to reduce the WCRT and critical paths through each local tick. Because ForeCast computes the WCET of each tick, a distribution of the tick execution times could be shown (e.g., as a timing profile of the program). This would help the programmer gauge the overall time predictability and performance of their system.

A ForeC and ForeMC debugger that supports the usual debugging facilities such as breakpoints, monitor memory locations and their values, observe the execution of each thread, and query other runtime information. A simulator could be included to help the programmer step through the execution of their program in an interactive manner. The simulator would show how the threads execute in parallel and how the values of (private and shared) variables change during execution. Similar to counter-examples in model checking, the simulator could show a runtime scenario that leads to the program’s WCRT. Such a simulator would be invaluable in the debugging of timing violations.

9.2 Final Remarks

Cyber-physical systems are expected to become more intricate, but simplicity in the development process will aid their success. The ForeC language, ForeMC framework, and ForeCast analyzer form an integrated tool chain that can simplify the formidable challenges of future CPS development. ForeC offers a compelling alternative to the writing of parallel programs instead of relying on C and third-party support or on the extraction of parallelism from traditional synchronous programs. ForeC marries the convenience of programming in C with the formalism of synchronous languages. ForeCast is a powerful analyzer that can compute tight WCRTs, limiting the over-provisioning of system resources. ForeMC expands the reach of ForeC and other synchronous languages from the safety-critical domain to the mixed-criticality domain. The efficacy of developing mixed-criticality CPSs was demonstrated by extensive benchmarking. Finally, the work in this thesis opens up many avenues that reach far into the future.
Published


Under Preparation


This appendix describes how the combine policies and combine functions work together to combine more than two copies of a shared variable. We compare the behaviors of the combine policies, all, new, and mod, using an illustrative example. We also provide additional examples of combine functions for primitive C data types and for programmer-specified data structures.

B.1 Combining More Than Two Copies

The ForeC program shown in Figure B.1a is used to explain how multiple copies of a shared variable are combined. The program’s control-flow graph is shown in Figure B.1b. The program has a shared variable called s and uses the combine function plus. The pre of s is 3 for the program’s first tick. Figure B.1c shows the copies of s at the end of the first tick, organized by the thread genealogy. Each node represents a thread and its copy, e.g., [main: 3] means that the main thread has a copy with the value 3. Copies that were assigned a value during the tick have the • symbol, e.g., [tA: 1•] means that thread tA’s copy has been assigned the value 1. A thread without a copy has an empty value, e.g., [tB:] means that thread tB does not have a copy. Arrows are drawn from the child threads to their parents to show the thread genealogy. Note that thread tC creates its copy from tA’s copy because it exists (see Chapter 3.1.4). Hence, the value of thread tC’s copy is 1 rather than 3. By contrast, thread tE creates its copy from the pre of s because thread tB does not have a copy of s.

The formal semantics of ForeC presented in Chapter 3.3 defines how more than two copies of a shared variable are combined. Specifically, the copies are combined with the semantic function Combine (see Chapter 3.3.3) in the (par-4), (par-5), (par-6), and (par-7) rules for the par statement. To summarize, the copies from sibling threads (i.e., threads forked by the same par statement) are combined and
shared int s=3 combine with plus;
void main(void) {
    read(s);
    par(tA(), tB());
}
void tA(void) {
    s = 1;
    par(tC(), tD());
}
void tC(void) { read(s); pause; }
void tD(void) { s = 2; pause; }
void tB(void) {
    par(tE(), tF());
}
void tE(void) { read(s); pause; }
void tF(void) { s = 5; pause; }
int plus(int th1, int th2, int pre) {
    return (th1+th2);
}

(a) ForeC program.

(b) Control-flow graph.

(c) The copies when tick 1 ends.

(d) Policy all.

(e) Policy new.

(f) Policy mod.

Figure B.1: Example of combining multiple copies of a shared variable.
their combined value is assigned to their parent thread. Then, the copies of the parent and its sibling are combined together and assigned to their parent. This continues until the main thread is reached. Figure B.1d illustrates this for the combine policy all, where all the copies are combined. The final combined value is 11 and is assigned to shared variable s to complete the resynchronization.

For the combine policy new, the copies that have the same value as pre are ignored. For the copies shown in Figure B.1c, thread main and tE’s copies of s would be ignored. Figure B.1e illustrates how the copies are combined for the combine policy new. Note that thread tF’s copy is assigned directly to tB because its sibling’s copy is ignored. The final combined value is 8.

For the combine policy mod, the copies that have not been assigned a value during the tick are ignored. For the copies shown in Figure B.1c, thread main, tC, and tE’s copies of s would be ignored. Figure B.1f illustrates how the copies are combined for the combine policy mod. The final combined value is 7.

B.2 Combine Policies Illustrated

This section illustrates the behavior of the combine policies all, new, and mod by using the example of Figure B.2. Figure B.2a shows a block diagram of the ForeC program shown in Figure B.2b. In each tick of the program, the number of times that the two buttons (button1 and button2) are pressed is outputted. On line 6 in Figure B.2b, threads t1 and t2 are forked to test whether each button has been pressed. The results are assigned to the shared variable count. Line 6 also forks thread t3 to read the value of count and output it to display. Hence, three copies of count will be created in each tick of the program. The copies of count are combined with the function plus (line 18). Table B.1a provides possible inputs for five ticks of the program and the expected values for the pre (resynchronized value) of count. For example, because both buttons are pressed in tick 3, we expect the pre of count to be 2 in the following tick.

For each thread, Table B.1b shows their value for count and the value of pre that is calculated when the combine policy is all. Recall that the policy all combines all the copies. For tick 1, pre = 0, the initialization value of count. When tick 1 ends, plus(plus(0, 0, 0), 0, 0) = 0. Thus, for tick 2, pre is 0. When tick 2 ends, plus(plus(1, 0, 0), 0, 0) = 1. Thus, for tick 3, pre = 1. For tick 4, pre = plus(plus(1, 1, 1), 1, 1) = 3. This is unexpected because pre should be 2. The problem occurs because thread t3’s copy is being summed with t1 and t2’s copies. Ideally, we are only interested in summing t1 and t2’s copies.
188 Shared Variables

![Diagram of Button counter]

(a) Task.

```c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

void main(void) {
  par(par(t1(), t2()), t3());
}

void t1(void) {
  while (1) { count = (button1==1); pause; }
}

void t2(void) {
  while (1) { count = (button2==1); pause; }
}

void t3(void) {
  while (1) { display = count; pause; }
}

int plus(int th1, int th2, int pre) { return (th1+th2); }
```

(b) ForeC program.

Figure B.2: Example of counting the number of button inputs.

<table>
<thead>
<tr>
<th>Tick</th>
<th>button1</th>
<th>button2</th>
<th>pre of count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tick</th>
<th>count</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>t1's copy</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>t2's copy</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>t3's copy</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>pre</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

(a) Inputs and expected pre of count.

<table>
<thead>
<tr>
<th>Tick</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tick</th>
<th>t1's copy</th>
<th>t2's copy</th>
<th>t3's copy</th>
<th>pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Combine policy all.

<table>
<thead>
<tr>
<th>Tick</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tick</th>
<th>t1's copy</th>
<th>t2's copy</th>
<th>t3's copy</th>
<th>pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) Combine policy new.

<table>
<thead>
<tr>
<th>Tick</th>
<th>t1's copy</th>
<th>t2's copy</th>
<th>t3's copy</th>
<th>pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(d) Combine policy mod.

Table B.1: The pre of count for each combine policy.
We continue to get unexpected values for \( \text{pre} \) for the remaining ticks. For tick 5, \( \text{pre} = \text{plus}(\text{plus}(1, 0, 3), 3, 3) = 4 \). For tick 6, \( \text{pre} = \text{plus}(\text{plus}(0, 0, 4), 4, 4) = 4 \).

We can try and fix the situation by changing the combine policy of \( \text{count} \) to \( \text{new} \). For each thread, Table B.1c shows their value for \( \text{count} \) and the value of \( \text{pre} \) that is calculated when the combine policy is \( \text{new} \). Recall that the policy \( \text{new} \) ignores the copies that have the same value as \( \text{pre} \). For tick 1, \( \text{pre} = 0 \), the initialization value of \( \text{count} \). When tick 1 ends, all the copies are ignored. Thus, for tick 2, \( \text{pre} \) remains at 0. When tick 2 ends, thread \( t_2 \) and \( t_3 \)’s copies are ignored. Thus, for tick 3, \( \text{pre} = 1 \). When tick 3 ends, all the copies are ignored. Thus, for tick 4, \( \text{pre} \) remains at 1. This is unexpected because \( \text{pre} \) should be 2. The problem occurs because the result of testing whether a button has been pressed happens to be the same value as \( \text{pre} \). When tick 4 ends, thread \( t_1 \) and \( t_3 \)’s copies are ignored and \( t_2 \)’s copy is assigned directly to \( \text{count} \). Thus, for tick 5, \( \text{pre} = 0 \) and is an unexpected value. For tick 6, \( \text{pre} \) remains at 0.

We can correct the situation by using the combine policy \( \text{mod} \). For each thread, Table B.1d shows their value for \( \text{count} \) and the value of \( \text{pre} \) that is calculated when the combine policy is \( \text{mod} \). Recall that the policy \( \text{mod} \) ignores the copies that have not been assigned a value during the tick. Thus, by inspecting Figure B.2b, we can see that thread \( t_3 \)’s copies will always be ignored because it is never assigned a value. In contrast, thread \( t_1 \) and \( t_2 \)’s copies will be always be assigned a value and, therefore, be combined in each tick. Thus, the values of \( \text{pre} \) in Table B.1d matches with the expected values shown in Table B.1a. Note that, for other programs, it is possible that the combine policies \( \text{all} \) or \( \text{new} \) may be more appropriate.

### B.3 Examples of Combine Functions

We begin by presenting combine functions that are based on associative and commutative mathematical operators. These combine functions are shown in Figure B.3. Examples of their behaviors are given after each function as comments. Since the functions do not use the value of \( \text{pre} \), its value is represented by a “?”. Combine functions can use the \( \text{pre} \) of a shared variable to perform computations that relate to past values. For example, Figure B.4a is a ForeC program with two threads that assign the number of items they have processed to their copy of the shared variable \( \text{count} \). The goal of the combine function \( \text{total} \) is to compute a running total of the number of items processed by the threads. The function sums the values of both copies and \( \text{pre} \) (the previous running total). An example of its behavior for two ticks is provided after the program as comments. However,
the combine function behaves incorrectly when more than two copies need to be combined. This is because the value of `pre` would be summed multiple times, equal to the number of times that the combine function is invoked. Figure B.4b is a corrected version of the program. Both threads now increment their copy of `count` with their own number of processed items. Thus, each thread creates their own running total. The combine function `total` now sums the running totals of both copies, which results in `pre` (the previous running total) to be summed twice. Thus, `pre` is subtracted from the sum. An example of its behavior for two ticks is provided after the program as comments. The value of each copy is shown as `a+b`, where `a` is the `pre` of `count` and `b` is the value returned by `item()`.

Combine functions can also be defined for programmer-defined data structures. For example, Figure B.5a defines a C-struct called `ProdSum` that stores the product (`prod`) and sum (`sum`) of the numbers assigned to it. The combine function `prodsum` multiplies all the values in `prod` and sums all the values in `sum`. An example of its behavior is provided after the function as comments.
B.3 Examples of Combine Functions

shared int count=0 combine all with total;
par(f(),f());
}
void f(void)
{
while (1) {count=items(); pause;}
}
int total(int th1,int th2,int pre) {return (pre+th1+th2);}
// Tick 1: total(1,2,0)=3
// Tick 2: total(3,4,3)=10
// The expected total: 1+2+3+4=10

(a) Running total for two values.

shared int count=0 combine all with total;
par(g()),par(g(),g())
}
void g(void)
{
while (1) {count=count+items(); pause;}
}
int total(int th1,int th2,int pre) {return (th1+th2+pre);}
// Tick 1: total(0+1,total(0+2,0+3,0),0)=total(0+1,5,0)=6
// Tick 2: total(6+4,total(6+5,6+6,6),6)=total(6+4,17,6)=21
// The expected total: 1+2+3+4+5+6=21

(b) Running total for more than two values.

typedef struct {
  int prod;
  int sum;
} ProdSum;

ProdSum prodsun(ProdSum th1,ProdSum th2,ProdSum pre) {
  th1.prod=th1.prod*th2.prod;
  th1.sum=th1.sum+th2.sum;
  return th1;
}
// th1={.prod=2,.sum=2} and th2={.prod=5,.sum=5}
// minmax(th1,th2,?)={.prod=10,.sum=7}

(a) Product and sum of two values.

typedef struct {
  int value;
  int min;
  int max;
} MinMax;

MinMax minmax(MinMax th1,MinMax th2,MinMax pre) {
  th1.min = min(th1.value,th2.value);
  th1.max = max(th1.value,th2.value);
  return th1;
}
// th1={.value=2,.min=0,.max=0} and th2={.value=5,.min=0,.max=0}
// minmax(th1,th2,?)={.value=2,.min=2,.max=5}

(b) Minimum and maximum of two values.

Figure B.4: Examples of combine functions that use pre.

Figure B.5: Examples of combine functions for C-structs.
For another example, line 1 of Figure B.5b defines a C-struct called `MinMax` that stores an assigned value (`value`), and the minimum and maximum values (`min` and `max`) that have been assigned to `value`. The idea is to create a shared variable using `MinMax` and have threads make assignments to `MinMax.value`. The combine function (line 3) computes the minimum and maximum values that have been assigned to `value`. An example of its behavior is provided after the function as comments. Indeed, the combine function will only behave in an associative and commutative manner if the threads only write to `value` and not read from it.
References


References


