http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

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

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

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

To request permissions please use the Feedback form on our webpage. http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
Architectures Specific
Compilation for Efficient
Execution of Esterel

Simon Yuan
July 2013

Supervisors: Dr. Partha S. Roop
Prof. Zoran Salcic

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

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

Software for embedded systems has traditionally been written in languages such as C. However, C does not provide primitives for describing concurrency and typically requires an operating system (OS) for emulating concurrency. The combined effect of scheduling in the OS and the lack of semantics introduce non-deterministic behaviour to the system. Subsequently, when problems arise, they are often difficult to reproduce and debug.

The synchronous programming paradigm offers a refreshing approach to specification of system level design. The formal semantics of synchronous programming languages established a well defined behaviour of the primitives provided by them such that the behaviour of any implementation of these languages strictly follow their semantics. As a result, both the high level specifications and their implementation are closely associated.

Esterel is one of such synchronous languages, featuring imperative style syntax, native concurrency, preemption and exception. While Esterel offers powerful features, compiling Esterel has been challenging. The conventional approaches to Esterel compilation for software require implementing an Esterel program with complex and low level control-flow using C. The overhead from such implementations often rivals the actual control code of Esterel. Alternatively, Esterel programs can be translated to digital circuits as the synchronous nature of Esterel maps well to the behaviour of logic gates. This hardware approach offers the best performance at the cost of flexibility compared to the software approach.

This thesis seeks to address these problems with a novel intermediate approach by compiling Esterel for hardware architectures tailored for execution of Esterel. To overcome the limitations of the software and hardware implementation of Esterel, we introduce an execution platform that accelerates the execution of Esterel with a set of Esterel-oriented instructions for handling concurrency and preemption in hardware. Experimental results have shown significant reduction in generated code size while gaining speedups compared to the pure software approach.

Another aspect of compilation of Esterel that has been relatively unexplored is the implementation of its concurrency with true parallelism on multi-core architectures. There have been minimal attempts to address compilation of Esterel for multi-cores due to the
challenges of preserving Esterel semantics with true parallelism and gain speedups against sequential execution.

To address the problem of compiling Esterel for multi-core, we have adopted the dynamic scheduling technique such that the problem scheduling and load distribution can be addressed independently. This thesis covers a static approach and a dynamic approach for distributed execution on multi-cores. Experimental results have shown that the speedups gained from static distribution are highly dependent on the amount of data code in Esterel, whereas speedups are achieved with both control and data-dominated Esterel programs using the dynamic distribution approach.
Acknowledgements

The completion of thesis would not have been possible without the contributions from many people. First among them is my supervisor, Partha S Roop, by whom I was given the opportunity to take on a Ph.D. If it was not for him, I would not have the privilege of working with him. I am truly grateful for his guidance and encouragement, especially during my most difficult times.

I would also like to acknowledge Zoran Salcic who also supervised me in this thesis. His input during the early stage of this research has been invaluable.

I thank Li Hsien Yoong who has been an important mentor throughout the course of this Ph.D. Many inspirations of the work presented in this thesis had originated from countless technical discussions with him.

I also appreciate contributions from Roopak Sinha whose professional advice on formal proofs has been essential.

Finally, I would like to acknowledge Avinash Malik, Sidharta Andalam and Cheng Pang, who have added an entertaining aspect to the experience of doing a Ph.D.
# Contents

1 Introduction ........................................ 1
   1.1 Preliminaries of Esterel ............................. 8
      1.1.1 Compiling Esterel ............................... 10
   1.2 Research contributions ............................. 12
   1.3 Thesis organization ............................... 13

2 The Esterel Language .................................. 15
   2.1 Reactive systems ............................... 16
   2.2 The synchronous programming paradigm .............. 17
   2.3 The Esterel language ............................... 18
   2.4 Syntax and semantics of Esterel ...................... 20
      2.4.1 The strong abort statement .................. 20
      2.4.2 The strong abort immediate statement ......... 21
      2.4.3 The weak abort statement .................... 21
      2.4.4 The weak abort immediate statement .......... 22
      2.4.5 Handling data ............................ 23
      2.4.6 Causality issues ........................... 23
      2.4.7 Schizophrenic behaviour .................... 24
   2.5 Discussion ........................................ 24

3 Literature Review .................................... 25
   3.1 Compiling Esterel for sequential execution .......... 26
   3.2 Compiling Esterel for distributed execution ........ 28
   3.3 Architectures for reactive systems ................. 29
   3.4 Intermediate formats ................................ 30
      3.4.1 The unrolled concurrent control flow graph .... 33
      3.4.2 The GRC intermediate format ................ 34
   3.5 Discussion ........................................ 37
## 4 Architecture for Efficient Execution of Esterel

4.1 Architecture of the STARPro processor ........................................... 40
   4.1.1 The datapath ................................................................. 41
   4.1.2 The Thread Control Block (TCB) ........................................ 41
   4.1.3 The Abort Handling Block (AHB) ........................................ 44
4.2 The instruction set architecture ......................................................... 49
4.3 Execution semantics ........................................................................ 53
4.4 Experimental results ...................................................................... 56
4.5 Discussion ...................................................................................... 62

## 5 Compiling Esterel for the STARPro Architecture

5.1 The intermediate format ................................................................. 65
   5.1.1 The hierarchical state graph ................................................. 67
   5.1.2 The unrolled concurrent control-flow graph ............................ 68
   5.1.3 Handling synchronous preemption ......................................... 83
   5.1.4 Translation example ............................................................. 98
5.2 Scheduling .................................................................................... 101
   5.2.1 Clustering ........................................................................ 101
   5.2.2 Priority assignment ............................................................ 102
   5.2.3 Inserting *context switch* nodes ........................................ 108
5.3 Handling schizophrenic programs ...................................................... 109
5.4 Discussion ...................................................................................... 111

## 6 Compiling Esterel for Multi-core Execution

6.1 Run-time signal resolution ............................................................. 115
6.2 Implementation of run-time signal resolution ................................... 119
   6.2.1 The GRC intermediate format ............................................. 119
   6.2.2 The insertion of signal resolution nodes ............................... 121
   6.2.3 Generating code with run-time signal resolution ................. 126
6.3 Static load distribution .................................................................. 131
   6.3.1 The static load distribution algorithm ................................ 131
   6.3.2 Generating code from GRC ............................................... 133
6.4 Dynamic load distribution .............................................................. 138
   6.4.1 High level representation of dynamic thread distribution ........ 140
   6.4.2 Implementation of dynamic thread distribution .................. 143
6.5 Correctness of run-time signal resolution ........................................ 147
   6.5.1 Base case ....................................................................... 149
   6.5.2 Hypothesis ...................................................................... 149
   6.5.3 Inductive step .................................................................. 149
6.6 Experimental results ......................................................... 150
  6.6.1 OS-based implementation using POSIX threads ................. 151
  6.6.2 Using an embedded multicore architecture ....................... 152
  6.6.3 Performance of static load distribution ......................... 154
  6.6.4 The load balance of static load distribution ................... 154
  6.6.5 Performance benchmark of dynamic load distribution .......... 156
6.7 Discussion ..................................................................... 157

7 Conclusions and Future Work .............................................. 161
  7.1 Future work ................................................................ 163
    7.1.1 Handling data computation .................................. 163
    7.1.2 Fine-tuning compilation for multi-core ....................... 164
    7.1.3 Handling larger class of Esterel programs ................... 164
  7.2 Final remarks ................................................................ 165

A List of Publications ........................................................... 167

References ........................................................................... 169
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The high level representation of the ABRO example in SyncChart [1]</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>An example of ABRO implemented in C</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>An example implementation of ABRO in C++</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>The example implementation of ABRO in Esterel</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>An example execution trace of the ABRO example</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>Illustration of interleaving between threads in order to satisfy dependencies</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>The schizocyc example Esterel program</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of a strong abort: (a) Esterel source; (b) Reaction timeline</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>An example of a strong-immediate abort: (a) Esterel source; (b) Reaction timeline</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>An example of a weak abort with a loop: (a) Esterel source; (b) Reaction timeline</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>An example of a weak-immediate abort: (a) Esterel source; (b) Reaction timeline</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>The ParallelData example written in Esterel</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>A reaction timeline of the ParallelData example</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>The ParallelData example represented in UCCFG</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>The ParallelData example represented in the Graph Code Format</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>The STARPro architecture: (a) Overview of hardware blocks; (b) Thread Handling Block (TCB); and (c) Abort Handling Block</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>The internal view of the datapath</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>The interfacing between the AHB and the TCB</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Finite State Machine of the Abort Handling Block: (a) The FSM; (b) State transition conditions</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>The SchizoCyc example Esterel program</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>The SchizoCyc example translated to STARPro assembly</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.7 Change of thread context for SchizoCyc example (a) assembly (b) change of thread context ........................................ 54
4.8 Comparison of hardware resource usage between KEP3a and STARPro . . 57
4.9 Performance (in run-time machine instruction count) comparison in bar graph (the lower the better) ........................................ 60
4.10 Code size comparison in bar graph (the lower the better) ................. 61

5.1 Unrolled Concurrent Control-Flow Graph of the SchizoCyc example .... 66
5.2 Hierarchical state graph nodes ........................................... 68
5.3 Unrolled concurrent control graph nodes .................................. 69
5.4 Mapping of $p = q; r$ .................................................. 72
5.5 Mapping of $p = q||r$ .................................................. 73
5.6 Mapping of $p = \text{emit } S$ ............................................. 73
5.7 Mapping of $p = \text{pause}$ ................................................. 73
5.8 Mapping of $p = \text{present } S \text{ then } q \text{ else } r \text{ end}$ ................. 74
5.9 Mapping of $p = \text{loop } q \text{ end}$ ......................................... 75
5.10 Mapping of $p = \text{signal } S \text{ in } q \text{ end}$ ................................... 76
5.11 Mapping of $p = \text{await } S \text{ do } q \text{ end}$ ................................ 76
5.12 Mapping of $p = \text{await immediate } S \text{ do } q \text{ end}$ .................... 77
5.13 Mapping of $p = \text{await } S \text{ do } q \text{ case immediate } T \text{ do } r \text{ end}$ .... 78
5.14 Mapping of $p = \text{suspend } q \text{ when immediate } S$ .................... 79
5.15 Mapping of $p = \text{suspend } q \text{ when } S$ ................................ 79
5.16 Mapping of $p = \text{abort } q \text{ when } S \text{ do } r \text{ end}$ ...................... 80
5.17 Mapping of $p = \text{abort } q \text{ when immediate } S \text{ do } r \text{ end}$ ........... 82
5.18 Tracking the tick boundaries using the HSG ............................. 83
5.19 Mapping of a strong abort: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline ........................................ 85
5.20 Mapping of a strong-immediate abort: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline ........................................ 87
5.21 Mapping of a weak-immediate abort: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline ........................................ 88
5.22 Mapping of a weak abort with a loop: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline ........................................ 90
5.23 Mapping of a strong abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$ ................................. 91
5.24 Mapping of a strong-immediate abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$ ................................. 92
5.25 Mapping of a weak-immediate abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$ ................................. 92
5.26 Mapping of a weak abort with a loop using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFGsd .................................................. 93
5.27 Mapping of a suspend: (a) Esterel source; (b) Assembly; (c) UCCFGsd; (d) Reaction timeline .................................................. 95
5.28 Mapping of an immediate suspend: (a) Esterel source; (b) Assembly; (c) UCCFGsd; (d) Reaction timeline .......................... 96
5.29 Mapping of a trap in UCCFGsd .................................................. 97
5.30 The SchizoCyc example Esterel program .................................. 98
5.31 Visual representation of the Abstract Syntax Tree ....................... 99
5.32 Translation process of the SchizoCyc example .......................... 100
5.33 The clustering algorithm ...................................................... 103
5.34 Nodes of the SchizoCyc example grouped into clusters ............... 104
5.35 The priority assignment algorithm ........................................ 106
5.36 Tracing upwards from a join node ........................................ 107
5.37 The context switch node insertion algorithm .......................... 109
5.38 Completed UCCFGsd of the SchizoCyc example ......................... 110
6.1 A cyclic executive for run-time signal resolution ........................ 116
6.2 An example of run-time signal resolution .................................. 117
6.3 The ParallelData example written in Esterel .......................... 120
6.4 The ParallelData example represented in the Graph Code Format ...... 121
6.5 Illustration of places to insert resolution nodes .......................... 122
6.6 Algorithm for inserting signal resolution nodes ........................ 123
6.7 Auxiliary functions for the insertion algorithm .......................... 124
6.8 An example of (a) a valid and (b) false dependency .................... 125
6.9 Algorithm for removing false dependencies ............................. 126
6.10 An sketch of an example Esterel program ................................ 127
6.11 The GRC representation of the sketch example ........................ 128
6.12 The generated code of the sketch example with run-time signal resolution .................................................. 129
6.13 The partitioned GRC of the ParallelData example ..................... 132
6.14 The distribution algorithm ................................................... 134
6.15 The partitioned GRC of the sketch example ............................. 135
6.16 The generated code of the partitioned sketch example ................ 136
6.17 The reentrant version of the high level cyclic executive with run-time signal resolution .................................................. 139
6.18 The GRC representation of the sketch example ........................ 141
6.19 The time-line of the state of the thread queue and processors .......... 142
6.20 The sketch example illustrating thread queue access in the generated code .................................................. 145
6.21 Content of the (a) thread queue; and (b) mutex .......................... 146
6.22 A cyclic executive for run-time signal resolution ................. 148
6.23 Performance comparison on PC (the lower the better) ............ 152
6.24 Memory architecture of a quad Microblaze system: (a) the memory architecture; (b) the memory address space ................. 153
6.25 Performance comparison on dual-core Microblaze (lower the better) .... 155
6.26 ACRT comparison of LZSS only on dual-core Microblaze (lower better) . 155
6.27 Speedup of the average case reaction time ....................... 157
6.28 Speedup of the worst case reaction time ......................... 158
List of Tables

2.1 List of pure Esterel kernel statements ........................................... 20
2.2 List of Esterel statements for interfacing with the host language .... 23

4.1 Esterel-oriented instructions .......................................................... 49
4.2 Hardware resource usage – STARPro vs KEP3a ................................. 57
4.3 A list of example Esterel programs used .......................................... 58
4.4 The worst and average case reaction time ....................................... 58
4.5 Performance (in run-time machine instruction count) ....................... 59
4.6 Code size comparison using different compilation techniques ............ 60
4.7 Effectiveness of pipelining in clock cycles per instruction ................. 61

6.1 A list of benchmarking programs ................................................... 150
6.2 Performance comparison on PC ..................................................... 152
6.3 Comparison of processor idle time in % ....................................... 156
The standard practice of designing software for embedded systems follows the top-down approach starting with abstract models of the system. This model driven engineering approach is both intuitive for human to understand and implement. Specification of embedded systems requires high level modeling to capture control-flow, data-flow and concurrency. Conventional tools typically involve modeling of the system using visual languages to aid the design of embedded systems.

While the control-flow, data-flow and concurrency are well described by high level models, implementing such abstract models is often a manual process. Subsequently, the links between the abstract models and their implementations are lost. These drawbacks of the conventional approach directly affect the development time, maintainability, robustness, and cost of a project. With the ever growing complexity of embedded systems, a better approach must be sought after.

For example, the common practice of implementing concurrency involves emulating concurrent tasks using an operating system (OS). Then, the actual control logic is typically implemented in C. The lack of semantics in such implementation means the behaviour of the program is implementation dependent on the OS and the C compiler. When problems arise, the high level specification cannot be matched to the implementation due to lack of semantics between the specification and its implementation. As a result, debugging the program has to be done at lower level and slows down the rate of development. The inconsistency between implementations of OSes and compilers also makes writing
safety critical embedded software rather difficult and time consuming. This is equivalent to the days of writing software at low level using the assembly language before high level languages were invented. To illustrate this, let us try implement a variant of a simple example called ABRO, in C. ABRO, a simple program developed by Berry [2], may be considered as the *hello world* of synchronous languages.

![ABRO diagram](image)

Figure 1.1: The high level representation of the ABRO example in SyncChart [1]

**ABRO** is a simple program that awaits the inputs A and B. The program will emit O to the output when both A and B are received by the program. During the execution of the program, its state can be reset if the input R is received. The behaviour of the program can be represented using a special kind of state machine called SyncChart in Fig. 1.1.

Each rounded rectangle in Fig. 1.1 represents a hierarchical state. A hierarchy of states can be embedded inside a state, such as the **ABRO**, **ABO** and **AB** states. The **ABO** state has a loopback transition from itself when R is received. The small red dot at on the starting point of the arrow indicates that the transition is forcibly taken when R is received. Inside the **ABO** state, the **AB** state is entered initially whenever the **ABO** state is entered. The **AB** state is further divided into two concurrent state machines demarcated by the dashed horizontal line. The A state and the B state await for their input signals independently, but must exit the **AB** state jointly when both of the concurrent state machines reach their terminal states. When both input A and B are received, the program takes the transition from the **AB** state to the terminal state in the **ABO** state and emits O.

Fig. 1.1 succinctly captures the behaviour of the **ABRO** example with concurrent state machines. If we were to draw a conventional state machine without concurrency, the **AB**
state would have to be flattened into a cross-product of the two concurrent state machines. The number of states would be exponential to the number of concurrent state machines involved. However, concurrency is not native to typical languages used for programming embedded systems, such as C and C++. Let us now implement the ABRO example in C first using the POSIX thread API. An example implementation is depicted in Fig. 1.2.

Let us assume input and output signals can be accessed by the interface functions provided in interface.h. The interface functions, named by the signal name prefixed with input or output, interacts with the environment. For example, inputA() reads the A input, while outputO() emits to the O output. The taskA() and taskB() functions have been added to the program to add some imaginary computation. Moreover, threads communicate with each other like tasks in an embedded system.

An inspection of the implementation would reveal that the program revolves around two threads. The state machine is explicitly described by the switch statement inside the infinite loop in main(). The initial state on line 24 creates two threads by passing the pointers to threadA() and threadB() to pthread_create(). The POSIX thread objects pthreadA and pthreadB are initialized by pthread_create(), and are used later as a way to refer to the two threads. The AB state on line 29 continuously checks for R. If R is received, the state machine is reset to the initial state and the two threads are terminated synchronously by lines 31∼34. If R is not received, the status of the two threads are checked on line 36. When the two threads are ready to join, O is emitted and the program moves on to the terminal state on line 42. Finally, the program stays in the terminal state until R is received, and the program is reset to the initial state.

The state machine of the two threads is implemented implicitly by the while statements on lines 6 and 13. As soon as a thread receives the input signal it has been waiting for, it performs some computation by calling taskA() for threadA, or taskB() for threadB. The results from taskA() and taskB() are communicated across to the other thread. Depending on the results, X and Y may be emitted.

There are a few problems with the implementation, both syntax-wise and behaviourally. We will list some of the problems below:

1. Multi-threading is done through library calls, requiring explicit effort to manage threads. This is both counter-intuitive and error-prone as there is no abstraction over multi-threading.

2. There is no hierarchical state control. Instead, the transition condition is checked in individual states. For example, the input R is checked in both the AB state and the terminal state. This further adds more risk of making mistakes.

3. The behaviour of this multi-threaded program is highly implementation specific due to lack of semantics at the language level. The behaviour of the program is totally
```c
#include "interface.h"
extern bool taskA();
extern bool taskB();
volatile bool resultA, resultB;
void* threadA(void*) {
    while (!inputA());
    if (resultB)
        outputX();
    resultA = taskA();
    return NULL;
}
void* threadB(void*) {
    while (!inputB());
    resultB = taskB();
    if (resultA)
        outputY();
    return NULL;
}
int main() {
    pthread_t pthreadA, pthreadB;
    enum { AB_I, AB, AB_End } state = AB_I;
    while (true) {
        switch (state) {
        case AB_I:
            pthread_create(&pthreadA, NULL, threadA, NULL);
            pthread_create(&pthreadB, NULL, threadB, NULL);
            state = AB;
            break;
        case AB:
            if (inputR()) {
                pthread_cancel(threadA);
                pthread_cancel(threadB);
                pthread_join(&pthreadA, NULL);
                pthread_join(&pthreadB, NULL);
                state = AB_I;
            } else if (pthread_kill(pthreadA, 0) &&
                       pthread_kill(pthreadB, 0)) {
                state = AB_End;
                outputO();
            }
            break;
        case AB_End:
            if (inputR())
                state = AB_I;
            break;
        }
    } return 0;
}
```

Figure 1.2: An example of ABRO implemented in C
non-deterministic due to scheduling and non-portable. This is one of the main reasons a multi-threaded C program is notoriously known to be difficult to debug. The ambiguity comes from the implementation of the C compiler, the threading library, and finally the operating system.

4. Due to problem 3, there is no strict order of the communication between \texttt{threadA()} and \texttt{threadB()}. This ambiguity makes managing shared resources difficult and error-prone. In fact, accessing shared resources is not even atomic. A thread may modify some shared data while another thread reads the shared data at the same time. The result of the simultaneous access becomes undefined.

5. Due to problem 4, a programmer needs to explicitly protect shared resources with mutexes to ensure atomic access. This creates yet another opportunity for mistakes. For a complex program, it is not uncommon to find a program deadlocks due to a thread waiting indefinitely for other threads to release the access to the shared resource. It is also possible a thread becomes starved due to its inability to gain access to the shared resource.

Clearly, writing concurrent programs in C is difficult. Let us try another attempt to implement ABRO in C++. An example of the implementation in C++ is presented in Fig. 1.3.

The C++ implementation abstracts out the threading part by introducing the \texttt{Thread} class. \texttt{ThreadA} and \texttt{ThreadB} are then implemented as sub-classes of the \texttt{Thread} class. As a typical implementation of a thread class, a thread is expected to define its behaviour by reimplementing the \texttt{run()} method. After creating an instance of the thread, the thread can be started by calling the \texttt{start()} method, which spawns the thread and calls \texttt{run()}. The thread stops when either \texttt{run()} returns or the instance is destroyed.

Syntactically, the C++ implementation provides some abstractions over the C implementation with thread objects. The structure of the program remains largely the same. With the example implementation, the problems of concurrent programming in C still mostly applies to C++, only problem 1 is slightly overcome. However, the example implementation can be improved by providing some abstraction over the state-machine with state-machine objects. Implementing the state-machine with objects can hide the control of the state-machine away. The state-machine object can then be event driven stimulated by the environment inputs. This would solve problem 2, but not the remaining problems. The problem with threads in conventional programming languages is well described in [3].

Over the years, researchers have been developing new languages to provide higher level abstractions in an attempt to address the shortcomings of the conventional programming languages. In particular, one class of such languages has received significant attention – the family of synchronous languages. Synchronous languages such as Esterel [4], Lustre
```cpp
#include "interface.h"
extern bool taskA();
extern bool taskB();
volatile bool resultA, resultB;
class ThreadA : public Thread {
  protected:
    void run() {
      while (!inputA());
      if (resultB)
        outputX();
      resultA = taskA();
    }
};
class ThreadB : public Thread {
  protected:
    void run() {
      while (!inputB());
      resultB = taskB();
      if (resultA)
        outputY();
    }
};

int main() {
  ThreadA *threadA, *threadB;
  enum { AB_Init, AB, AB_End } state = AB_Init;
  while (true) {
    switch (state) {
    case AB_Init:
      threadA = new ThreadA;
      threadB = new ThreadB;
      threadA.start();
      threadB.start();
      state = AB;
      break;
    case AB:
      if (inputR()) {
        delete threadA;
        delete threadB;
        state = AB_Init;
      } else if (threadA.isFinished() && threadB.isFinished()) {
        delete threadA;
        delete threadB;
        state = AB_End;
        outputO();
      }
      break;
    case AB_End:
      if (inputR())
        state = AB_Init;
      break;
    }
  return 0;
}
```

Figure 1.3: An example implementation of ABRO in C++
module ABRO_Data:
input A, B, R;
output X, Y, O;
function taskA() : boolean;
function taskB() : boolean;
signal resultA, resultB in
loop
  await A;
  present resultB then
    emit X;
  end;
  if taskA() then
    emit resultA;
  end
  ||
  await B;
  if taskB() then
    emit resultB;
  end;
  present resultA else
    emit Y;
  end
end signal
emit O;
each R
end signal
end module

Figure 1.4: The example implementation of ABRO in Esterel

[5], Signal [6], offer a refreshing alternative for developing embedded software with their high level design primitives. The key advantage of synchronous languages is that the formal semantics of these languages allows programs written in these languages to be verified in a mathematically sound manner. This is a big contrast to the traditional software testing and debugging approach. Instead of extensive simulation and testing, properties of the software behaviour can be formally proved using static analysis.

Out of the aforementioned synchronous languages, Esterel, in particular, is an imperative language. The imperative programming paradigm is what most embedded software programmers are familiar with. Let us take a look at how the ABRO example is implemented in Esterel. An example implementation is presented in Fig. 1.4.

The most noticeable difference compared to the implementation in C and C++ is the succinct and clean syntax. Esterel supports multi-threading natively with its parallel operator (||). Based on the structure of the program and keywords used, the syntax of the
language provides ample hints as to how the program might behave. Due to the semantics of Esterel, implementing the ABRO example in Esterel not only describes ABRO precisely, the implementation has none of the problems listed earlier. This is because Esterel guarantees that all operations are atomic. For example, the status of signals resultA and resultB are guaranteed to be known by the time these signals are read. The subsequent section will provide a quick overview of Esterel using the example in Fig. 1.4.

1.1 Preliminaries of Esterel

The programming language Esterel [2] belongs to the family of synchronous languages [7]. Due to its synchronous semantics, all correct Esterel programs are guaranteed to be reactive and deterministic [8]. These properties greatly simplify the formal verification of programs, while providing predictable run-time behaviour. Hence, there has been a great deal of interest in using Esterel for the design and validation of a special class of embedded systems, called reactive systems [9].

Esterel provides constructs to describe concurrently executing statements. Each concurrent component executes in lock-step, evolving in discrete instants of time, known as a tick. Such synchronous execution is achieved by taking a snapshot of input signals at the start of each tick, performing some computation, and emitting all outputs before the start of the next tick. Concurrent statements may communicate back and forth with each other within a tick, making such communication conceptually instantaneous. Such synchronous execution guarantees that each reaction in Esterel is atomic in every possible sense. This makes race conditions, common in concurrent programming, impossible in Esterel.

While such powerful features make it intuitive to write specifications in Esterel, its compilation and efficient execution has been non-trivial. Some aspects of this complexity can be illustrated using the example shown in Fig. 1.4.

An Esterel program always consists of basic entities called modules. The example in Fig. 1.4 has only a single module named ABRO. Within this module, a loop encloses the entire program, where the loop is restarted each time input signal R is present, except on the first tick. This preemtting behaviour of the loop-each construct is said to be a strong preemption. When the preemption is taken, its enclosed body is terminated immediately. The ‘∥’ operator inside on line 17 forks two threads. The threads communicate with each other through local signal resultA and resultB, declared on line 7. At the start of each thread, an await statement stops the thread from proceeding ahead until the awaited signal becomes present. These await statements correspond to the A state and the B state in Fig.1.1.

Communication between threads takes place immediately below the await statements. The present statements on lines 22 and 11 check whether signals A and B are present.
The emit statement inside the present statement is executed if the signal is present. The emitted signal instantly becomes present, and its presence is immediately detected by the corresponding present statement. The emission of resultA and resultB are determined by the returned result of the functions taskA and taskB, respectively. Note that between the two threads, the order of communication to the other thread and calling the function is opposite to each other. This is done to interleave the communication such that the threads would not deadlock due to cyclic dependencies [8]. In order to proceed to line 26, the two threads must both terminate synchronously. They will not join if only one thread finishes.

To illustrate the behaviour of the program, an example execution trace of the program is presented in Fig. 1.5. On the first tick, all input signals are present. As the await statement and the loop-each statement have an implicit one-tick delay, the program does not react to any of the inputs during the first instant. The control flow pauses on lines 6 and 13 of each thread, respectively. During the second tick, A becomes present, the first thread proceeds ahead from the await A statement. As B is still absent, the second thread continues to wait till the third tick. During the third tick, B becomes present, and the second thread moves on to join with the first thread. Both threads terminate synchronously and O gets emitted. In order to reset the program, R is activated on the fourth tick. During the fifth tick, all input signals become present again at the same time. The program takes the preemption, not allowing the two threads any chance to execute. Hence, O is not emitted. The last tick is the most interesting, but also the most complex with respect to the program flow in this example. When A and B are present at the same time, taskA and taskB execute concurrently. Assuming taskA returns false and taskB returns true, both X and Y are emitted in the same tick.

<table>
<thead>
<tr>
<th>ticks</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>R</td>
<td></td>
<td>X</td>
<td>Y</td>
<td>O</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.5: An example execution trace of the ABRO example

To understand how this happened, the micro-step execution of the program needs to be explained. Fig. 1.6 shows the interleaving between the two concurrent threads during the last tick such that their dependencies are satisfied. These threads are divided into three levels demarcated by the horizontal lines. At each level, a thread executes while the other waits due to a dependency. In Fig. 1.6, the second thread starts first as present resultB in the first thread depends on emit resultB in the second thread. When taskB returns true, resultB is emitted. Its presence is instantly picked up by the first thread, resulting
if taskB() then
    emit resultB;
end

if taskA() then
    emit resultA;
end

end

Figure 1.6: Illustration of interleaving between threads in order to satisfy dependencies

in X being emitted. The first thread then executes taskA, which happens to returns false and resultA is not emitted. The absence of resultA is also immediately picked up by the second thread, resulting in Y being emitted. Finally, when both threads finish, 0 is emitted. The immediate presence of a signal when emitted is called instantaneous communication. It is one of the features that makes Esterel powerful, but also difficult to compile.

The current generation of Esterel compilers are not very efficient for small embedded controllers. The generated code is both large in size and slow to execute. This is not surprising given the complexity of implementing a compiler that has to correctly preserve the semantics of Esterel. Moreover, with the recent advancement in processor technologies, even embedded systems have started using multi-core processors. However, the compiling techniques implemented in current Esterel compilers do not support generating code for parallel execution on multi-core. In fact, many compilers cannot even be extended to support parallel execution due to the underlying compilation techniques used.

1.1.1 Compiling Esterel

The ideal synchronous model of the language places significant effort on the compiler designers. The main effort in compiling Esterel is the issue of mapping the synchronous concurrency to an actual implementation. The conventional approach is to compile away the concurrency. Threads are first topologically sorted [10] based on the dependency between threads followed by sequentializing the threads in order to produce a statically scheduled sequential program.

The ABRO example demonstrates many aspects of the challenges of compiling Esterel. Most notably:

Instantaneous communication: all signals that are set to be present must remain
1.1 Preliminaries of Esterel

present throughout the current tick, and its status must be detected by all of its
testers.

**Cyclic dependencies:** threads communicate back and forth, forming a dependency cy-
cle between them. Some of these are legal (causal [8]) while others are rejected
(non-causal [8]).

**Detecting absence of signals:** non-emission of signals that are shared between threads.

Instantaneous communication is tricky to implement because the compiler must make
sure that the emitted signals are immediately readable by the signal testers. Traditionally,
Esterel compilers implement instantaneous communication by ensuring signal producers
are always executed before their consumers. The compilation process is further compli-
cated by cyclic dependencies. A thread may emit some signal to another thread, which
in turn replies back through another signal, forming a dependency cycle between the pair
of threads. `ABRO_Data` is such an example. The second thread checks for the `resultA`
followed by emitting `resultB`, while the first thread emits `resultA` followed by testing
of `resultB`. All these potentially happening in the same tick. Clearly, scheduling any of
these two threads before the other in a fixed order is simply wrong. The dependency is
in fact dynamically influenced by the stimuli from the environment.

For some cyclic Esterel programs, the dependency cycle exists only at the thread level.
Often when the program is analyzed in micro-steps within a single tick, the dependency
chain no longer forms a cycle. To compile such a program like the `ABRO_Data` example, the
compiler must interleave the threads such that the order of the communication always
happen in a fixed order. Typically, interleaving is achieved by breaking threads into
segments of code. Then a rearrangement of the segments is performed based on their
topological order such that the dependencies are satisfied.

Detecting the absence of a signal is more tricky than its presence. Presence of a signal
can be determined as soon as it is emitted while its absence require confirmation from
all statements that may potentially emit the signal such that they collectively agree on
the absence of the signal. Combining these challenges, compiling Esterel correctly while
producing efficient code is non-trivial.

There are two conventional approaches for compiling Esterel:

1. Translate an Esterel program into an equivalent digital circuit.

2. Compile away the concurrency within an Esterel program and sequentialize the
   program. The resulting control-flow is then described in C.

Approach 1 gives the best performance compared to the approach 2. A pure control
Esterel program is translated into hardware logic gates by converting the program into an
equivalent digital circuit [8]. The synchronous nature of the language closely resembles the clock synchronization in digital circuits. Mapping a pure control Esterel program without any data computation to equivalent logic gates is more straightforward than software implementation. While the best performance can be achieved with this approach, it is severely limited in flexibility. The design of hardware can no longer change once deployed whereas software can still be updated. Thus, producing hardware is also more costly than distributing software.

Approach 2 is much more flexible at the cost of performance compared to the first approach. The overhead is mainly due to the constant rearrangement of the way threads are interleaved. During each \textit{tick}, a different interleaving is being worked out for the following \textit{tick}. For simple control software, this constant reordering of thread segments may potentially create significant overhead relative to the actual Esterel code and requires a longer reaction time to environment inputs.

Interestingly, there is a third approach that is relatively less explored. The intermediate approach adopted by this research and prior research [11, 12], attempt to strike the right balance between the software and the hardware approach by providing hardware acceleration for language features that impact the performance. For example, handling thread management and preemption can potentially lower the overhead of executing Esterel programs. Moreover, with the advent of affordable multi-core microprocessors, compiling Esterel for such architectures must also be addressed. This thesis will introduce two types of processor architectures:

1. An architecture providing an instruction set specifically tailored for compactness and efficient execution of Esterel.

2. A multi-core architecture tailor designed for distributed execution of Esterel.

Both of these architectures will be complemented by compilers that take advantage of such architectures using various novel compilation techniques. The next section will highlight the significance of this research.

1.2 Research contributions

This thesis addresses the issues of both the efficiency of the generated code from Esterel, and the resulting code size. It introduces a processor tailored for efficient execution of Esterel, and a compiler specifically designed for this processor. This thesis also addresses the issues of compiling Esterel for execution in parallel on multi-core architectures. These architecture specific compilation techniques, together, cover the requirements for most applications of Esterel.
The intermediate approach to more efficient execution of Esterel is addressed through the following contributions:

**A novel Esterel tailored hardware architecture** that accelerates the execution of Esterel. This thesis introduces a processor architecture called STARPro designed to speedup the execution of Esterel by managing threads and preemption in the hardware. A set of Esterel oriented instructions has been designed to allow an Esterel program described with a similar syntax to Esterel.

**An architecture specific compiler** optimized to generate efficient code with STARPro’s Esterel oriented instructions. The compiler introduces a novel intermediate format that preserves the original structure of the Esterel source, while low level enough to allow direct mapping to STARPro assembly instructions. These specialized instructions speed up the execution of Esterel while reducing the generated code size significantly compared to conventional approaches (see Chapter 4 for details).

**Parallel execution on multi-core architectures** by resolving the status of signals at run-time as opposed to the conventional static approach. The run-time signal resolution approach not only enables parallel execution of Esterel threads, but makes load distribution far more flexible. A preliminary load distribution algorithm has been developed as an experiment to investigating the load balance of the distributed execution. The algorithm analyzes the size of each thread at compile-time as an approximation of the load. This initial load distribution approach helped identifying the limitations of distributing load statically. Consequently, a new dynamic load balancing approach that distributes load more evenly, resulting in significant speedups compared to sequential execution.

These contributions solve many aspects of the challenges involved in implementing Esterel. These include generating very compact code without sacrificing performance and generating code that takes advantage of multi-core architectures. The techniques for achieving these objectives will be described in the chapters outlined in the next section.

## 1.3 Thesis organization

This thesis is organized into six remaining chapters. Chapter 2 gives an overview of Esterel, covering its syntax and semantics. A sound understanding of Esterel is a prerequisite of the subsequent chapters.

Chapter 3 introduces the related work to this thesis. The current generation of Esterel compilers are covered in greater detail. The existing intermediate approaches to executing
Esterel are reviewed. The chapter also looks at previous work on the distributed execution of Esterel, the hardware architectures used, as well as their compilers.

Chapter 4 describes the hardware design of STARPro, its instruction set architecture, and the execution semantics. This chapter concludes with extensive benchmarks to evaluate the performance and code size of STARPro compared to pure software Esterel compilers and another Esterel tailored hardware architecture.

Chapter 5 continues to describe the compiler developed for STARPro. This chapter introduces the intermediate format used by the compiler and how each Esterel statement is represented by the intermediate format. Then, the scheduling algorithm is presented, followed by an example of translating Esterel to the intermediate format.

Chapter 6 covers the run-time signal resolution approach followed by the proof of the correctness of the approach. The chapter then goes on to present both the static and dynamic load distribution. This chapter concludes with exhaustive benchmarking to evaluate both the performance benefit of parallel execution on multi-core architectures and the load balance. Finally, the thesis is concluded with closing remarks and potential areas for future work.
Chapter 1 provided a quick preview of Esterel. In this chapter, we seek to describe the notion of reactive systems, the synchronous programming paradigm for describing such systems, and a more thorough overview of Esterel.

Development of Esterel started in the early 1980’s, evolving over time with semantical and syntactical changes. Since then, research interest around Esterel has picked up by the time Esterel reached V5. Several Esterel V5 compilers have since been developed. In more recent years, Esterel V7 has extended V5 with broad syntactic changes and data handling capabilities. Despite those changes, the semantics of Esterel V7 remain largely the same as V5. With literature and resources of Esterel V5 widely available compared to V7, it makes more sense to base this research on Esterel V5. In fact, the proprietary Esterel Studio [13] is the only Esterel V7 compiler available before its development and support ended after the intellectual property had been acquired by Syfora.

The remainder of the chapter will give a brief introduction to the Esterel syntax and its semantics with a few examples. This chapter will not attempt to describe the language in full detail. Instead, the reader is referred to the Esterel Primer [2] for the full features, and [8] for the formal semantics.
2.1 Reactive systems

The notion of reactive systems and transformational systems were originally introduced in [9]. An embedded system is classified as a reactive system if the system must react to the environment at the speed determined by the environment. A digital wrist watch is an example of a reactive system. In contrast, a transformational system actively prompts to its environment for inputs. Then, the input is transformed into output. Electronic calculators fall into this category.

Later, Berry [2, 14] introduced a third classification for interactive systems, where the system interacts with its environment through a master. The master will only respond to input events at its convenience when possible. An operating system is a prime example of such systems. These three classes of systems have different characteristics that target different applications. Reactive systems emphasize prompt reaction to input events with correctness and timeliness of the outputs. André et al. [15] highlighted the following requirements reactive systems need to meet:

**Safety:** Reactive systems designed for mission critical applications demand guarantees about their behaviour with respect to safety requirements. Such safety critical systems can be found in automatic flight control and nuclear power plants controllers.

**Temporal:** Given an input, the temporal correctness with respect to outputs of a reactive system is equally required in addition to its logical correctness. In order to satisfy this requirement, the specification of the system must place a bound on the length of time the system takes to respond to an input event.

**Concurrency:** Describing complex systems with sets of modules co-operating to achieve the specified behaviour is both intuitive and convenient to humans. The concurrent behaviour of such systems are typically specified by orthogonal states, or implemented as parallel composition of modules [9].

**Determinism:** Expectations of reactive systems typically mandate consistent behaviour when given the same set of inputs repeatedly. This makes a deterministic system simpler to specify and debug compared to a non-deterministic system [16].

For some reactive systems, their emphasis is on the control of the outputs based on their inputs. These systems are categorized as control-dominated systems, such as human-machine interfaces, like washing machines and coffee makers. Some reactive systems react to streams of input regularly and produce outputs continuously. These systems fall into the category of data-dominated systems, such as navigation devices and modern flat-screen televisions.
2.2 The synchronous programming paradigm

Esterel specializes in describing control-dominated reactive systems [2] with its synchronous semantics, allowing formal verification of both the logical and temporal behaviour of the program. These are some of the characteristics of the synchronous programming paradigm described in the next section.

2.2 The synchronous programming paradigm

In the past, transformational systems dominated the development of programming techniques [2]. As a result, procedural languages became mainstream at the time and the techniques quickly became mature. However, the focus on transformational systems has largely neglected concurrency and event-handling capabilities within these languages.

As the needs to deal with interactive systems grew, techniques have been developed to fulfill these needs. This has lead to the rise of operating systems and distributed algorithms for concurrent programming. Concurrent programming has also inspired new languages such as ADA [17] and OCCAM [18] to support concurrency natively. These languages introduce higher level of abstraction with language constructs for asynchronous concurrency. However, the asynchronous model of concurrency is non-deterministic. Berry and Gonthier [16] point out the following issues with these classical concurrent languages:

- Reactions compete for resources, creating race conditions. Atomicity of a reaction had to be explicitly ensured by the programmer.
- Lack of timing specification at the language level, requiring priorities to be used to provide timing guarantees.
- Inconsistent view of the environment from the perspective of sub-processes in a program due to asynchronous read operations. The same input may return different values when sampled at different times.

Due to these difficulties, the classical concurrent languages are not well suited for programming reactive systems. Synchronous concurrent languages, in contrast, are the right tools for these purposes. The synchronous programming paradigm assumes an infinitely fast reactive system that is able to react to its inputs synchronously. This assumption, called the synchronous hypothesis, is the key to how the synchronous approach works. With such assumption, each reaction is treated as being instantaneous, and thus atomic in any possible sense. Instructions can then be divided into instantaneous and non-instantaneous categories. An instantaneous instruction would execute without incrementing the system clock, whereas a non-instantaneous instruction would cause a delay.

Though, the synchronous hypothesis only holds if the system is able to execute fast enough to keep up with the rate of the incoming input events. When the hypothesis
is satisfied, the system can be treated as a sequence of discrete instants with nothing happening between the completion of the current one until the start of the next. Thus, an event can be said to be reacted to by concurrent reactions if they execute in the same instant when the event occurred. The idealized timing preciseness simplifies reasoning about the temporal behaviour and removes non-determinism from communication taking place asynchronously.

The deterministic nature of the synchronous languages, such as Esterel, is what makes the approach well suited for programming reactive systems. The underlying model of Esterel can be summarized with the following features:

Reactive and atomic reactions: An Esterel module can be viewed as a black box that requires activation from the outside in order to produce a reaction. The reaction completes within an instant of logical time, called a tick. As reactions are bounded between ticks, they cannot overlap, ensuring the atomicity of each reaction.

Instantaneous broadcast: Communication by default is instantaneous in Esterel, allowing concurrent statements within the same tick to share the same synchronous view.

Synchronous preemption: When preemption primitives in Esterel are applied to concurrent statement, those statements controlled by the preemption will always react to the preemption on the same tick. This allows precise behavioural and temporal control over concurrent statements.

Almost all Esterel programs make use of these features, as we will see with a few example Esterel programs in the next section.

2.3 The Esterel language

Earlier in Chapter 1, we briefly introduced Esterel without describing in detail. This section will describe a different example in full. Presented in Fig. 2.1 is a single module named SchizoCyc. The start of a module usually defines the interface the module interacts to the environment with. The input, output, and inputoutput keywords in SchizoCyc declare inputs R and I, outputs C and D, and a port for both input and output named B. Within this module, an abort construct encloses the entire program, where input signal R can preempt the abort body on any tick, except at the starting instant (strong preemption). In the absence of the preempting signal, the program executes continuously in an endless loop. The ‘∥’ operator inside the loop forks two threads within the loop. The threads communicate with each other through local signal A and output signal C.
2.3 The Esterel language

```plaintext
module SchizoCyc:
  input I, R;
  output C, D;
  inputoutput B;
  abort
  loop
    signal A in
    present A then emit B end;
    pause;
    present B then emit A end;
    await A;
    emit C;
    ||
    await immediate I;
    emit A;
    present C then emit D end;
  end signal
end loop
when R
end module
```

Figure 2.1: The schizocyc example Esterel program

The first thread checks for the presence of signal A in the starting instant, and emits B if A is present. Similarly in the following instant, if B is present, A is emitted, otherwise the program waits until A becomes present before emitting C. The two signal presence and emission statements within this thread form a cyclic producer-consumer dependency [8]. It is permitted in this example because the dependency cycle is broken across instants by the pause statement. On each iteration of the loop, signal A will be reincarnated [19]. This is equivalent to unrolling the loop body to produce a new declaration of the local signal A for each pass of the loop.

The second thread waits for input I to become present. As soon as I becomes present, A is emitted immediately. Following the emission of A, if C is emitted by the first thread, D will also be emitted in the same instant. Because of signal reincarnation, the example in Fig. 2.1 is said to be schizophrenic. When I is present in the first instant, signal A can be both present and absent in the same instant. Hence, SchizoCyc is named after the schizophrenic behaviour, and a cyclic dependency. The SchizoCyc example illustrates some of the aspects that make the compilation and efficient execution of Esterel challenging. Correct scheduling of the threads such that the data dependencies between the threads are satisfied, contributes significantly to this complexity. Moreover, preserving the synchronous semantics of Esterel in compiled code is already, in itself, non-trivial.
2.4 Syntax and semantics of Esterel

The examples described in the previous section presented some of the Esterel constructs. In this section, we will take a look at the syntax and semantics of the kernel statements of pure Esterel. A list of pure Esterel kernel statements is presented in Table 2.1. The kernel statements are the most basic constructs. The more high level constructs can be composed from the kernel statements. In the table, the symbols $p$ and $q$ represent composition of other Esterel statements. For a full list of Esterel statements, the reader is recommended to take a look at [8].

Table 2.1: List of pure Esterel kernel statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nothing</td>
<td>a null statement that does nothing</td>
</tr>
<tr>
<td>pause</td>
<td>stops execution for one tick</td>
</tr>
<tr>
<td>$p ; q$</td>
<td>executes $p$ followed by $q$ in sequence</td>
</tr>
<tr>
<td>$p \parallel q$</td>
<td>executes $p$ and $q$ concurrently</td>
</tr>
<tr>
<td>loop $p$ end</td>
<td>executes $p$ forever</td>
</tr>
<tr>
<td>signal $S$ in $p$ end</td>
<td>declares signal $S$ in $p$</td>
</tr>
<tr>
<td>emit $S$</td>
<td>emits signal $S$</td>
</tr>
<tr>
<td>present $S$ then $p$ else $q$ end</td>
<td>executes $p$ if $S$ is present; otherwise $q$</td>
</tr>
<tr>
<td>suspend $p$ when $S$</td>
<td>freezes execution of $p$ for as long as $S$ is present</td>
</tr>
<tr>
<td>trap $T$ in $p$ end</td>
<td>declares an exception $T$ in $p$ for catching it</td>
</tr>
<tr>
<td>exit $T$</td>
<td>throws an exception $T$</td>
</tr>
</tbody>
</table>

In Esterel, instantaneous loops are not allowed. That is, the $p$ statement enclosed in the loop has to include at least one statement that introduces a delay of at least one tick, such as a pause statement. Delaying is one of the ways in Esterel to control timing. The sensitivity in terms of timing of some statements can also be altered, such as the await and the abort statements. The abort statement, in particular, offers four ways to control the timing of the preemption. We will explain the subtle difference in behaviour of each type of abort over the next subsections.

2.4.1 The strong abort statement

The strong abort statement is a preemption that terminates its body as soon as the preemption signal becomes present, except in the first tick. This behaviour can be illustrated through the example shown in Fig. 2.2.

The example shows an abort enclosing three emit statements divided into separate ticks to illustrate the effect of the preemption on its body. An example execution trace is shown in Fig. 2.2(b). The preemption signal – $S$ is present in the first two ticks, but the first emit statement in the abort body is still allowed execute. Only in the second tick does the abort react to $S$. The abort terminates its body, followed by emitting $Y$. 
2.4 Syntax and semantics of Esterel

2.4.2 The strong abort immediate statement

The strong-immediate version is a minor variant of the abort statement. This type of preemption also reacts to the preemting signal in the first tick, as illustrated in Fig. 2.3(b). The first emit statement inside the abort body is immediately preempted on the first tick. The immediate keyword is used to declare such behaviour.

```
1 abort
2 emit X;
3 pause;
4 emit X;
5 pause;
6 emit X
7 when immediate S;
8 emit Y
```

Figure 2.3: An example of a strong-immediate abort: (a) Esterel source; (b) Reaction timeline

2.4.3 The weak abort statement

The weak abort statement differs to the strong version by offering its body one last chance to execute before the end of the tick. Like the strong version, the immediate keyword can also be applied to control whether it should react in the first tick.

```
1 abort
2 emit X;
3 pause;
4 emit X;
5 pause;
6 emit X
7 when S;
8 emit Y
```

Figure 2.4: An example of a weak abort: (a) Esterel source; (b) Reaction timeline
2.4.4 The weak abort immediate statement

Having seen three types of abort already, it is not difficult to guess what the weak-immediate version of abort would behave. Indeed, the preemption is able to take place on the first tick, and the body is still allowed to complete its tick, as illustrated in Fig. 2.5(b).
2.4 Syntax and semantics of Esterel

2.4.5 Handling data

Esterel provides very limited language primitives to handle data. Instead, Esterel relies on interfacing with a host language, typically C, to handle data. A list of statements for handling data is presented in Table 2.2. Creating custom data types are supported by the type statement and manipulated in the host language. Data objects cannot be directly shared between threads, but only through local signals.

To interface with the host language, three types of language bindings are provided. The first type is the function statement that declares a host function. A function in Esterel should not have any side-effect on the arguments passed to it. The function must return a value when it terminates. The second type is the procedure statement that declares a host procedure. Unlike a function, a procedure does not return anything. Instead, references to data objects can be passed to the procedure by listing data objects in the first pair of parenthesis. The second pair of parenthesis passes data objects by value. A procedure is called by the call statement. The third type is the task statement that declares a task. A task is syntactically similar to a procedure, except that the task is expected to be non-instantaneous. A task is started by the exec statement.

Table 2.2: List of Esterel statements for interfacing with the host language

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>declares a data type</td>
</tr>
<tr>
<td>function</td>
<td>declares a function</td>
</tr>
<tr>
<td>procedure</td>
<td>declares a procedure</td>
</tr>
<tr>
<td>call</td>
<td>calls a function</td>
</tr>
<tr>
<td>task</td>
<td>declares a task</td>
</tr>
<tr>
<td>exec</td>
<td>executes a task</td>
</tr>
</tbody>
</table>

2.4.6 Causality issues

While instantaneous broadcast is a powerful feature of Esterel, it has the potential to produce non-deterministic code if not handled properly. The issue stems from instantaneous read-write cycles between signal readers and writers, such as the program

```
A else emit A end
```

The program reads, if A is absent, immediately emit A. The program contradicts with itself, as a signal cannot change its status within a tick. With complex structure of concurrent statements, this problem is more likely to occur. Such program would be non-causal and non-deterministic, and should ideally be picked up by the Esterel toolchain.
2.4.7 Schizophrenic behaviour

A program may become *schizophrenic* if a local signal is being read or written to, and re-declared in the same *tick*. Obviously, this can only happen when the local declaration of a signal is enclosed in a loop, like the program:

```plaintext
1 loop
2     signal S in
3       pause;
4       emit S
5     end
6 end loop.
```

When the loop restarts the local declaration, the signal from the previous iteration of the loop is *reincarnated* with a new initial value. The compiler would typically handle signal reincarnation with separate copies of $S$ to store the values of different incarnations.

2.5 Discussion

We have so far provided a tour through the basic features of Esterel through a few small examples to illustrate the intuitive syntax and programming style of Esterel. Esterel provides a set of language primitives to describe concurrency (′∥′), exception (trap) and preemption (abort). While these powerful features make Esterel intuitive to write specification for embedded systems, compiling Esterel has been non-trivial.

One of the most challenging aspect of compiling Esterel is the complexity of implementing its synchronous concurrency. The challenges stem from the tightly coupled nature of threads in Esterel with frequent communication back-and-forth between them. The compiler must correctly schedule these tightly coupled threads such that the semantics of instantaneous broadcast is adhered to. The next chapter will provide a brief review of the existing Esterel compilation techniques and their implementations.
The previous chapters have introduced Esterel V5 [16], as well as challenges of compilation of Esterel. This chapter will survey some of the existing work related to the compilation and the execution of Esterel.

Several approaches exist for dealing with these complexities in the compilation and execution of Esterel programs. These include hardware compilation [8], software compilation for general-purpose microprocessors [20, 21, 10], and architecture-specific compilation for reactive processors optimized for Esterel [11, 12]. While the translation of Esterel to digital circuits in hardware is relatively straightforward, the generation of efficient software code has been challenging. Software compilers typically map Esterel programs into another language, such as C, so that they can be executed on standard microprocessors. Consequently, concurrent statements in Esterel need to be interleaved and appropriately scheduled in order to produce an equivalent sequential program. This requires additional synchronization mechanisms to be added to preserve Esterel’s semantics. Such mechanisms introduce extra execution overhead and increase the required memory footprint.

The issues of the existing software implementation of Esterel will be highlighted in Sections 3.1 and 3.2. Following that, Section 3.3 will review some of the architectures designed for reactive systems and their support for execution of Esterel. Section 3.4 will describe the intermediate formats used in various Esterel compilers. These intermediate formats have heavily influenced the design of the compilers created for this research. This overview of the intermediate formats lays the foundation for the internal details of the
compilers created for this research that will be discussed in the next few chapters. Finally, Section 3.5 will conclude the chapter with some remarks on the motivation behind the research undertaken for this thesis.

3.1 Compiling Esterel for sequential execution

There are several approaches to implement Esterel, the two main ones are hardware implementation and pure software implementation. The former approach translates an Esterel description into hardware description that can be synthesized into a digital circuit. The latter approach uses various techniques to compile away the concurrency of Esterel into sequential software code, typically C. The compiled Esterel code is typically implemented as a reactive function that gets called each tick. Then, the reactive function interacts with its environment through a wrapper function, such as:

```c
int main() {
    for (;;) {
        sampleInputs();
        exitCode = reactiveFunction();
        processOutputs();
    }
}
```

This section will briefly describe the techniques developed for Esterel compilers taking the latter approach.

One of the first Esterel compilers is Berry et al’s V3 [16]. The V3 compiler represents the entire concurrent Esterel program with a single flattened automaton. The automaton is constructed by simulating Gonthier’s IC (intermediate code) format. The IC format consists of two parts: a concurrent control flow graph (CCFG) that hangs off a reconstruction tree. The construction tree is responsible for resuming a tick, handling concurrency and preemption as the program progresses. Executing the code produced from the automaton is very fast, but at the cost of code size. The automaton grows exponentially to the size of the original source, making this approach impractical for large programs.

The V3 compiler is later superseded by the V5 compiler with a totally different approach. The V5 compiler constructs a netlist of Boolean logic gates that are topologically sorted. Then the compiler generates code to simulate the behaviour of each gate. The resulting code size scales better than V3 as the number of gates generated is much more linear with respect to the original source. Like a real circuit, the program has to simulate the whole circuit in every tick, including the inactive statements in that tick. The result is significant slower performance compared to V3 and the more recent Esterel compilers [22, 10, 21, 20].
3.1 Compiling Esterel for sequential execution

The newer generation of the compilers achieved further improvement over the earlier generations with respect to performance and code size. Compilers such as Closse et al’s SAXO-RT [20], Edwards’s CEC [10], and Potop-Butucaru’s grc2c [21] represent an Esterel program with some directed graph that captures the control and data dependencies between the nodes. Once Esterel statements are translated into a graph, concurrent branches are topologically sorted and interleaved into a sequential order. The difference between these compilers are highlighted below:

**SAXO-RT** generates an event graph from the source followed by flattening of the graph into a sequential execution structure. The nodes in the event graph are grouped into *basic blocks* that are topologically sorted and selectively *enabled* and *disabled* by the scheduler at run-time.

**grc2c** relies on an intermediate format called the GRC format. GRC succinctly represents the complex nesting of flow control statements with clever encoding that allows generating very compact code from it. The speed of the generated code is comparable to that of automata-based compilers, and the advantage of similar, or even smaller code size than the netlist-based compilers. More details of the GRC format will be described in section 3.4.

**CEC** relies on a slight variant of GRC to generate code. The difference lies within the way control flow encoding is done. The main difference to grc2c is that CEC offers three different techniques to generate code. The compiler offers the option to either compile using a program dependency graph (PDG), dynamic list scheduling, or generate a virtual machine (VM). The PDG approach restructures GRC into the well known PDG representation [23]. Next, the compiler inserts test and guard variables into the PDG to implement the context switching between concurrent statements. The dynamic list approach is similar to how SAXO-RT works, except that in CEC, it only schedules those *basic blocks* that are participating during each *tick*. Unlike SAXO-RT, these non-participating *basic blocks* do not need to be explicitly *disabled*. The VM approach is an extension to the PDG approach by generating bytecode from the it. The only advantage it offers is saving in codes size. The VM approach runs slower than the other two. The PDG approach is slightly faster than the dynamic list approach in some cases, but not as good as the dynamic list approaches for programs with high level of concurrency.

Later on, Brandt et al. [24] introduced a compiler that supports separate compilation of a derivative language of Esterel, called Quartz. Quartz has a lot of similarities with Esterel that their compilation techniques are also applicable to Esterel. In that work, the intermediate representation of a Quartz program is broken into so-called jobs for each
control location corresponding to a tick boundary. These jobs are then saved in a file for linking at a later stage. When a module A instantiates another module B, the jobs within module B are extracted from the file for linking to module A. Finally, causality issues are checked at the end of the compilation process. There has been minimal work involving separate compilation of synchronous languages like Esterel due to the complication of handling causality [8] issues and schizophrenic [19] programs.

3.2 Compiling Esterel for distributed execution

The compilers mentioned in the previous section focus on sequential execution by compiling away the concurrency. However, the complexity of compiling Esterel for distributed execution received minimal attention. The main challenge stems from enforcing the order of communication dictated by the Esterel’s semantics, while communication is taking place in parallel.

Caspi et al. [25] introduced distributed execution of Esterel over an asynchronous network of processors by replicating Esterel control code on all processors, only distributing data. Distribution actually happens only after concurrency has been removed from the generated code. Each processor in the network is then assigned a portion of the signals in Esterel, but the statuses of the signals are made available to its peer processors through message passing. The main motivation of this work is to distribute code based on the physical location of sensors and actuators, rather than for performance gains.

More relevant is the work in [26], which described an approach to parallelize synchronous programs using the OpenMP [27] framework. That work was not targeted towards any one particular synchronous language, but for an intermediate format known as synchronous guarded actions. Groups of guarded actions with no dependencies can be found using an action dependency graph (ADG), and are marked as candidates for parallelization. The effectiveness of such an approach for Esterel is unclear, as the tight coupling between Esterel threads limits the parallelism that can be elicited from an ADG. In fact, [26] provides only a single benchmark to show speedup, and the benchmark itself is a data-dominated example.

Recently, Ju et al. [28] described an approach to estimate worst-case reaction time (WCRT) of Esterel programs running on multiprocessors. Interestingly, they adapted [29] for executing Esterel on a custom multi-core architecture, but did not report the estimated WCRT for multi-core execution in comparison to that for single-core execution on general-purpose processors.

Yoong et al. [30, 29] introduced a novel technique that schedule Esterel threads dynamically. The dynamic scheduling technique works on the principle that a signal is not read until its status is known. More details of this scheduling technique will be presented
in Chapter 6 where the dynamic scheduling technique is refined for execution on a multicores architecture. The compiler created in that work is able to generate code for two kinds of targets: sequential execution on a single-core or distributed execution on EMPEROR (see Section 3.3). EMPEROR is a multi-processor architecture tailored for execution of Esterel [11, 31]. While the compiler has been demonstrated to work correctly for sequential execution, compiling for distributed execution is somewhat untested. Later, a flaw in the implementation has been discovered while studying the dynamic scheduling technique in the research undertaken in this thesis. The flaw would under certain scenarios, incorrectly report the status of a signal as being known prematurely. Due to the flaw, an alternative implementation of the dynamic scheduling technique has been proposed in Chapter 6.

Cohen et al. [32] proposed an interesting technique to better parallelize computation using futures in Lustre. The technique allows a programmer to specify slow tasks to be computed asynchronously across ticks using the future construct. While the parallelization techniques we proposed are limited within tick boundaries, our focus is on parallelization of Esterel and load balancing with minimal user input.

In [33], Yip et al. explored static timing analysis on multicores by introducing a synchronous language based on C, called ForeC. Threads in ForeC, unlike Esterel, communicate through global shared variables. To ensure atomic access to shared variables, a copy of each shared variable is allocated for each thread that access it, and the resulting value of the variable is determined by a combine function associated with each shared variable. Such a communication mechanism greatly simplifies compilation process since there are no causality issues in ForeC. However, the question of load balancing is not directly addressed and relies on user input to indicate how threads should be distributed.

### 3.3 Architectures for reactive systems

The idea of designing reactive processors for reactive systems were explored by [34, 35, 36, 37] as an alternative approach for executing embedded software on standard microprocessors. These processors have been designed with instruction sets and architectural support for Esterel-like primitives. The instruction sets of these processors are capable of describing a reactive application succinctly, resulting in a more compact program and execute faster than equivalent implementations using standard microprocessors. These customized hardware, though, lack support for concurrency. They nevertheless have inspired later works, including this research, to design architectures with better support for execution of Esterel.

The first processor architecture that supports synchronous concurrency of Esterel was achieved by Dayaratne et al.’s EMPEROR [31, 11]. EMPEROR is a multi-processor ar-
chitecture that consists of multiple reactive cores. A reactive core is implemented by augmenting a PIC-like [38] microcontroller with reactive hardware units and a set of instructions for accessing the reactive hardware. To execute an Esterel program on EMPEROR, each thread is allocated to a reactive core. A core becomes active when its allocated thread is started by its corresponding Esterel ‘∥’ operator. The reactive cores execute in lock-step fashion using barrier synchronization. A hardware thread management unit controls the progression of each individual core with a bit vector representing the tick status of each core. When the corresponding flag in the bit vector of a reactive core is set, that reactive core is paused by the thread management hardware. As soon as the bit vector is unanimously set to ones by all active cores, a global tick is generated by the thread management unit to reset the bit vector and the reactive cores start executing synchronously. However, the one core per thread allocation scheme raises an issue. EMPEROR is not able to handle an Esterel program composed of more threads than the number of reactive cores. The thread allocation scheme also reduces the scalability of the architecture as supporting a large number of reactive cores is expensive with respect to hardware resources.

Later, Li et al. [12] introduced an Esterel processor that fully supports executing a pure Esterel program. The processor architecture provides an instruction set that resembles the high level Esterel constructs. The hardware is capable of handling synchronous concurrency, preemption, exception and communication using signals in Esterel. Supporting Esterel’s synchronous concurrency with hardware multi-threading is one of its novelties. Similar to traditional Esterel compilers, threads are divided into blocks of code and sorted based on their topological order. Each block is then assigned a priority that determines the order threads are scheduled by the thread management hardware unit at run-time. The multi-threading approach allows KEP3a to support a much larger number of threads than EMPEROR. The extensive support of Esterel in hardware, though, significantly increases the complexity of the hardware design.

Plummer et al. [39] have explored another approach of executing Esterel using a virtual machine (VM). The VM provides customized instructions to support Esterel’s execution, similar to KEP3a. The key difference is that a virtual machine is implemented as software, whereas KEP3a is a hardware platform. The code size in both approaches are superior when compared with traditional Esterel compilers. However, the VM approach is significantly slower than traditional Esterel compilers [39].

3.4 Intermediate formats

The previous chapters have provided an overview of the intuitive language constructs of Esterel and highlighted the complexities of handling such programming style by the com-
piler. Hence, to compile an Esterel program, an intermediate representation of an Esterel program is absolutely essential to facilitate analysis and translation of the language. This section will not provide an extensive survey of the various intermediate formats developed for the Esterel compilers discussed previously. Instead, the focus of this section is to introduce some of the intermediate formats in preparation for the compilation techniques discussed in the subsequent chapters. This section will describe an example Esterel program followed by two types of immediate representation in Section 3.4.1 and 3.4.2.

The *ParallelData* example shown in Fig. 3.1 is a parallel data processing pipeline consisting of three threads, demarcated by the || operator. These threads communicate via the local signals S and S2. The first thread begins by waiting for the Start signal. Once received, the program starts to process the data by calling the `processData1` procedure with result passed to the procedure by reference. The output of the procedure is then stored in result and sent using signal S to the second thread for further processing. The second thread waits for S using the await statement. Note that the data sent from the first thread is buffered and retrieved in the next tick using the `pre` keyword. This results in a software pipeline: while the second thread works on the data received from the first, the first thread continues to produce new data in parallel. As soon as the second thread completes its processing, the final result is sent to the third thread using the signal S2. If the CheckStatus input is activated, the body of the every statement in the third thread will start to execute. It takes the data from the second thread, does a self-diagnostic test with the given data, and then indicates whether the process is working normally. The program can be stopped at any time by activating the Stop signal.

In Fig. 3.2, no inputs were given in the first tick as ParallelData does not react to any input in the first tick. In the second tick, Start becomes present and detected by the await Start statement on line 11. The first thread enters the loop, calls processData1, emits S and finally stops at the pause statement on line 15. The second thread does not react to S as it reacts to the previous status of the S, which was absent in the first tick. The third thread remains awaiting for CheckStatus on line 24.

From the third tick onwards, processData2’s buffer is filled with data produced by processData1 in the first thread in the previous tick. As S became present in the second tick, the await `pre(S)` statement detects the presence of S, calls processData2 with the data embedded in S and finally emits S2. Both S and S2 are emitted in the third tick.

On the forth tick, CheckStatus becomes present. The first two threads continue to process the data and emit S and S2. The third thread detects CheckStatus and calls selfDiagnose on the data embedded in S2 on line 25. The selfDiagnose host function detects an error in the data, returns false and finally emits Error on line 28.

On the fifth tick, Stop becomes present and the Stop conditional node detects the signal and takes the present branch. Finally, the program terminates in this tick.
module ParallelData:
  input Start, CheckStatus, Stop;
  output Good, Error;
type Data;
procedure processData1(Data)();
procedure processData2(Data)(Data);
function selfDiagnose(Data) : boolean;
abort
signal S : Data, S2 : Data in
  var result : Data, final : Data in
  await Start;
  loop
    call processData1(result)();
    emit S(result);
    pause;
  end loop
  ||
  loop
    await pre(S);
    call processData2(final)(pre(?S));
    emit S2(final);
  end loop
  ||
every CheckStatus do
  if selfDiagnose(?S2) then
    emit Good;
  else
    emit Error;
  end if
end every
end var
end signal
when Stop
end module

Figure 3.1: The ParallelData example written in Esterel

ticks \rightarrow

<table>
<thead>
<tr>
<th></th>
<th>Start</th>
<th>CheckStatus</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>S2</td>
<td>Error</td>
</tr>
</tbody>
</table>

output

Figure 3.2: A reaction timeline of the ParallelData example
In the next two subsections, two types of intermediate representations of ParallelData will be presented.

### 3.4.1 The unrolled concurrent control flow graph

The unrolled concurrent control flow graph (UCCFG) is the intermediate format used by the EMPEROR’s compiler [31, 11]. The UCCFG representation describes an Esterel program with a set of nodes that closely resembles Esterel’s constructs. The types of nodes used to represent the ParallelData example are labeled and listed next to the graph in Fig. 3.3. The UCCFG representation feature the following nodes:

**Abort start and abort end**: These nodes are direct representation of the *abort-when* construct. The abort body is enclosed by these two nodes to confine the scope of preemption with a pair of *abort* nodes. The generated code from an *abort start* node initializes the reactive hardware in EMPEROR and starts monitoring the specified signal. If the preemption is taken during the execution of the abort body the reactive hardware sets the program counter register of the processor to continuation address pointed by the *abort end* node.

**Fork and join**: Concurrency of Esterel is preserved through these nodes. The *fork* node marks the start of concurrent execution paths while the *join* node merges those execution paths.

**Test**: The graphical representation of the *present* or the *if* statement.

**Emit**: The graphical representation of the *emit* statement.

**Await**: The graphical representation of the *await* statement. It is also inserted to explicitly mark *tick* boundaries by awaiting on a special signal called *tick*. The generated code from this node sets the flag in a bit vector that corresponds to the *tick* status of the thread, as described in Section 3.3.

**Action**: This node can represent various actions such as variable assignment and calling a procedure.

An Esterel program represented in UCCFG is both visually intuitive and compact thanks to the one-to-one mapping between some of the Esterel constructs and their representative nodes. The control-flow starts from the node at the top of the graph progresses through the graph by following the *control arcs* represented by the solid lines between nodes. The dependencies between *emit* nodes and *test* nodes are marked by the *data dependency arcs* that are drawn as dashed lines.
The entry point of the ParallelData example is the abort start node in Fig. 3.3. The abort nodes encloses the whole program and react to the Stop signal. The program creates three threads within the abort body with a pair of fork and join nodes. The first two threads, represented by the left branch and middle branch of the fork node, have a direct resemblance to the source. Both of these threads start with an await node followed by a procedure call and an emit node. There is no dependency between the emit S node and the test node in the second thread as the delayed reaction to S breaks the dependency. The every construct is translated into an await node, followed by a loop wrapped around a pair of abort nodes, which in turn encloses the body of the every construct followed by a loop around an await tick node. For details of thread scheduling and the translation process, they are described in [31, 11] and will not be covered here.

3.4.2 The GRC intermediate format

The graph code (GRC) format represents an Esterel program with an acyclic directed graph. It consists of two parts: (1) a hierarchical state graph (HSG), and (2) a concurrent control-flow graph (CCFG). As the discussions of the compilation techniques in this thesis do not involve the HSG, from here on, the term GRC will always be referred to the CCFG part of the representation.
The construction process of GRC introduces a unique distinction between the initial behaviour and the resumption behaviour of each Esterel statement, called the *surface* and *depth* behaviour respectively. We elaborate these terms with the following:

- The *surface* behaviour describes the micro-steps performed in the first *tick* of any composition of the kernel statements.

- The *depth* behaviour describes the micro-steps performed in the subsequent *ticks* of any composition of the kernel statements.

We will first describe the types of GRC nodes that exist in GRC followed by its control-flow. A list of GRC nodes is illustrated along the top of Fig. 3.4. These nodes are described as the following:

**Fork and Join**: Create and destroy threads respectively.

**Test**: The graphical representation of the present or the if statement.

**Action**: Performs an action such as variable assignment, emitting a signal and calling a procedure.

**Switch**: Encodes the state of a thread. Each branch under this node represents a tick.

**Enter**: Sets the state encoded in a switch node of a thread.

**Terminate**: This node, introduced in [10], is a variant of the sync node introduced in [21]. A terminate node encodes the completion status of a thread. A completion code of zero denotes a normal termination; a value of one indicates the thread has been paused for a tick; and a value greater than one indicates the thread has been terminated by an exception or preemption. This clever encoding scheme (courtesy of Berry [40]) succinctly captures the *meet at the rendezvous* behaviour of concurrent statements and the *winner-take-all* behaviour of simultaneously thrown exceptions and preemption. Within a pair of fork and join nodes, terminate nodes immediately precede the join node and dictate the continuation context the join node must take when threads terminate. This is the primary mechanism for compilation of nested exception and preemption. The hierarchical nesting of the exception and preemption constructs determine the priorities of their reactions when triggered simultaneously. The higher the priority the larger the value a terminate node returns. The highest value will take precedence when terminate nodes merge at a join node.

Like UCCFG, nodes in GRC are also connected by *control arcs* and *data dependency arcs* to describe the control-flow and communication between nodes respectively. The
biggest difference compared to the UCCFG representation is the acyclic control-flow in GRC. A GRC graph is traversed from top to bottom once every tick. There are no loops in GRC as the repetition is achieved by controlling of the states of the program in the subsequent tick using switch and enter nodes. Switch nodes are later translated into state variables that are assigned by enter nodes.

As an example, the GRC shown in Fig. 3.4 represents the ParallelData Esterel module. We will assume the same input trace as Fig. 3.2 to describe the control-flow. The program starts from the switch node at the top. The nodes residing on the left branch of the top switch node define the surface behaviour of the program. The program immediately pauses for one tick due to the await statements. Two enter nodes are inserted to setup the states of the subsequent tick by assigning $S_0 = 1$ and $S_1 = 1$.

In the second tick, the top switch node reads the value of $S_0$ and selects the middle branch. Start is now present and the program follows the present branch of the test Start node. $S_1$ is set to 0 to ensure that Start will no longer be tested in the subsequent ticks. Then, $S$ is emitted before the tick ends.

From the third tick onwards, processData2’s buffer, implicitly created by the pre operator, is filled with data produced by processData1 in the first thread in the previous tick. The pre(S2) test node in the second thread detects the presence of $S$ and selects the present branch. Then, S2 is emitted before the third tick ends. Both the first and the
second thread now repeat these execution paths indefinitely unless stopped by the **Stop** signal.

On the forth tick, **CheckStatus** becomes present. The first two threads continue to process the data and emit **S** and **S2**. The third thread detects **CheckStatus**, follows the present branch, then calls **selfDiagnose**. The procedure takes the data emitted via **S2** by the second thread and analyzes the data and discovers an anomaly. The thread takes the else branch of the **selfDiagnose conditional** node and emits **Error**.

On the fifth tick, **Stop** becomes present, which preempts the program due to the **abort when Stop** statements. The program follows the present branch and exits the program via the **enter** node at the bottom. This **enter** sets **S0 = 0** and forces the program to forever take the right branch of the top **switch** node in the subsequent **ticks**. This effectively terminates the program as the program will no longer react unless reset.

### 3.5 Discussion

This chapter has surveyed the pure software and the intermediate approach to compile and execute Esterel programs. Compiling Esterel with the pure software approach, in general, produces less efficient code in both performance and memory footprint compared to the intermediate approach. The intermediate approach to compile and execute Esterel, though, has only been explored by the works around EMPEROR [31, 11] and the KEP family of processors [36, 37, 41, 12]. There is room for improvement on the performance, memory footprint and hardware resources required for the Esterel tailored architecture, as exemplified by the techniques described in subsequent chapters.

There have also been minimal attempts at compiling Esterel for parallel execution on multi-core processors. The one thread per core approach taken by EMPEROR [31, 11] falls short on scalability as Esterel programs typically consist of a large number of threads. The thread distribution limitation on EMPEROR is somewhat improved by Yoong’s [30, 29] compiler. However, the problem of optimizing performance with better thread distribution was not a goal in that work. Moreover, the generated code did not take advantage of the hardware acceleration for Esterel provided by EMPEROR and merely treats EMPEROR as a network of generic microprocessors. Nevertheless, the compilation techniques for distributed execution of Esterel described in Chapter 6 has adapted Yoong’s novel scheduling technique.
This chapter will introduce an architecture specially designed to accelerate the execution of Esterel. This architecture-specific approach for Esterel execution relies on custom microprocessors that have been augmented with an instruction set, which enables efficient mapping of Esterel statements to assembly code. This approach yields very compact machine code, as well as efficient execution, and will be the focus of this chapter.

This chapter presents a novel multithreaded processor, named STARPro (Simultaneous multiThreaded Auckland Reactive Processor), that achieves significant speed-up and code size compaction over traditional methods for software implementations of Esterel. The proposed architecture is a fully pipelined, multithreaded, reactive processor called STARPro for direct execution of Esterel. STARPro provides native support for Esterel threads and their scheduling. In addition, it also natively supports Esterel’s pre-emption constructs, instructions for signal manipulation, and a notion of logical ticks for synchronous execution.

The EMPEROR multiprocessor architecture [11] was the first attempt at the direct execution of Esterel using a set of reactive processor cores. These cores communicate and synchronize with each other using a thread control block to achieve synchronous execution. It executes Esterel programs by resolving signal dependencies during run-time using a dual-rail encoding of signals [29]. This approach, while achieving good execution
times, required excessively high hardware resources.

In contrast to the approach taken in EMPEROR, new contributions were also made to the idea of reactive processing through the KEP series of processors [36, 37, 41, 12]. The KEP series of processors are custom designed architectures that have evolved from each generation with incremental support for executing Esterel. The most recent processor, KEP3a [12], is capable of preserving the semantics of the full language. It also provides a multithreaded execution platform to support the concurrency in Esterel. This approach has yielded impressive code size compaction and execution times, thus affirming again the benefits of reactive processors for executing Esterel.

However, there are many improvements that could be made over KEP’s approach to reactive processor design. At present, KEP3a employs a non-pipelined architecture, which supports Esterel’s semantics almost entirely in hardware. This approach results in a complex hardware design, with a consequently lower operating clock frequency.

In contrast, STARPro provides an alternative approach to direct execution compared to KEP3a. STARPro does not implement Esterel constructs in the hardware completely. Instead, it provides an infrastructure for efficient context switching and abort. It also relies on a pipelined architecture to obtain better average case performance compared to KEP3a. This has been achieved using far fewer logic gates for processor implementation, while maintaining code sizes that are slightly inferior to KEP3a.

This chapter is organized as follows. Section 4.1 presents STARPro’s architecture, which is followed by a description of its instruction set architecture (ISA) in Section 4.2. Section 4.3 describes how the ISA is used to express an Esterel program and its execution semantics. Section 4.4 shows the experimental results obtained for some benchmarks. Finally, Section 4.5 concludes the chapter with the discussion of the significance of the results.

4.1 Architecture of the STARPro processor

STARPro’s design extends a previous reactive architecture REMIC [35]. REMIC is a three-stage pipelined reactive processor that was inspired by Esterel, though it was not designed to provide support for executing Esterel. REMIC has a Reactive Functional Unit (RFU), attached to the control unit and datapath of the processor core, that provides instruction set support for efficient handling of asynchronous I/O in reactive applications. The RFU, however, is not well-suited for Esterel programs, which requires I/O to be handled synchronously. Moreover, REMIC has no support for concurrency. Hence, the Esterel Support Unit (ESU) has been developed to replace the RFU within REMIC, as illustrated in Fig. 4.1(a). The ESU still interfaces with the control unit and the datapath as before, but enables synchronous handling of signals, as well as multithreading.
to support concurrency in Esterel.

The ESU itself consists of the Abort Handling Block (AHB) for dealing with preemptions, and the Thread Control Block (TCB) for multithreading support. STARPro is not a typical SMT (simultaneous multithreading) processor [42] since it does not use a separate register file for each thread. Instead, it provides separate program counters and auxiliary registers for abort handling for each thread. In the following, how the datapath from REMIC is modified to work with TCB and AHB is first explained, before discussing how the two interact.

### 4.1.1 The datapath

STARPro is a RISC reactive processor, featuring a three-stage pipeline design. Its memory is organized following a Harvard architecture, with configurable access to either internal or external program memory and data memory. Both the program memory bus and the instruction width are 32-bit wide, while the data memory bus is 16-bit wide. I/O is mapped to the highest part of the address space. The datapath contains an 8-bank, 16-bit wide register file as general purpose registers. Next to the register file is the Arithmetic Logic Unit (ALU). It supports standard operations such as addition, subtraction, and bit shifting, just to name a few.

To support the additional needs of the ESU, the datapath of REMIC [35] has been extended with additional ports to the datapath external interface. The changes allow the ESU to directly access the register file, which now also serve as signal register. It allows any data loaded into the registers to be treated as signals. The other important modification adds a new input to the program counter multiplexer. This additional connection is required for loading a new program address when a preemption occurs. The next section will explain in more detail how the ESU uses these new connections to the datapath.

### 4.1.2 The Thread Control Block (TCB)

An Esterel program may have multiple threads. The TCB maintains the status of individual threads while emulating the synchronous concurrency using static thread scheduling. In Fig. 4.1(b), the TCB itself is composed of a scheduler that maintains thread context, a thread table, and a TCB control unit.

The thread table stores the current program counter and the abort context (the abort context will be described in Section 4.1.3) associated with the current thread. Both the program counter and the abort context are sufficient to fully describe a thread’s context in STARPro. The number of threads that can be stored in the thread table is parameterizable by design, and is limited only by the hardware resources available.

The thread table is indexed by the Thread ID register. The entry indexed by that
Figure 4.1: The STARPro architecture: (a) Overview of hardware blocks; (b) Thread Handling Block (TCB); and (c) Abort Handling Block
Figure 4.2: The internal view of the datapath
register determines the thread which is currently being executed. When the LD_TCB signal is asserted, write access is enabled to the table for a thread context to be saved. Switching between threads is achieved by changing the value stored in the Thread ID register. A new thread ID value is loaded through the Rx bus connected to the datapath. During the processor’s reset, the thread ID register will be initialized to zero. Consequently, the ID of the root thread of all programs will be assigned a default value of zero by the STARPro compiler.

The other important component of the TCB is the scheduler. The scheduler stores the priority and a notion of a local tick for each thread as boolean flags. We say that the local tick for a thread has elapsed whenever a pause statement is reached in that thread. This differs from the global tick for an entire Esterel program, which only elapses when all running threads have completed their local ticks. In STARPro, the pause statement is mapped to the PAUSE instruction, which is used within the processor to indicate the completion of the local tick for a given thread.

The scheduler will always select the thread with the highest priority for execution. In doing so, it ignores all the threads that have either completed their local ticks, or are otherwise inactive. A thread is considered to be inactive if its priority number is set to the lowest possible priority. When the local tick of all the currently active threads have elapsed, the global tick completes, and a compiler-generated management thread is selected to sample new inputs and to clear all outputs for the next global tick.

The distinction between local and global ticks is actually a key idea that facilitates the use of variable tick durations in STARPro. This idea was first introduced in [11]. By relying on the completion of individual local ticks to determine the final duration of a global tick, the global tick duration is dynamically changed and is equal to the actual computational time required for executing a number of threads in any instant.

4.1.3 The Abort Handling Block (AHB)

The AHB is used to monitor aborting signals, and to trigger the appropriate preemptions when necessary. In Esterel, the priority of the abort construct depends on the level of its nesting. An outer abort construct will always have higher priority over those nested below it. The AHB supports this feature by providing hardware-based priority resolution that is controlled by a finite state machine in the AHB. The depth of nested aborts is fully parameterizable by design. Fig. 4.1(c) depicts an AHB that has been configured with four levels of aborts for each thread. It is tightly coupled to the TCB, the interface between these two units are illustrated in Fig. 4.3.

The AHB relies on the abort context provided by the TCB to trigger abortions. An abort context consists of the following elements:
4.1 Architecture of the STARPro processor

Abort Handling Block (AHB)

- **Rx**: This is the bus that connects to a 16-bit register selected from the register file in the datapath. The register has to be loaded with the status of 16 I/O signals at a time from memory. It is updated at the beginning of every tick, and is used by the AHB to evaluate the current status of the aborting signals.

- **ASR<sub>n</sub>** (Abort Signal Register): This stores the ID of the signal which needs to be monitored during execution of an abort body. Each level of nested abort requires an additional ASR<sub>n</sub>, where the <i>n</i> indicates which level of the nested aborts it stores signal for. For example, if the hardware has support four levels, there will be ASR<sub>1</sub>~ASR<sub>4</sub>.

- **AAR<sub>n</sub>** (Abort Address Register): This stores the continuation address, to which the thread must jump to, should a preemption happen. The abort level is indicated by the <i>n</i>.

- **ATF<sub>n</sub>** (Abort Type Flags): STARPro supports the different types of abort in Esterel. Aborts can either be **strong** or **weak**, and may either be **immediate** or **non-immediate**. These are orthogonal to each other, resulting in four distinct behaviours for abortions in Esterel. The abort level is indicated by the <i>n</i>.

- **ALC** (Abort Level Count): Each thread may consist of an arbitrary number of nested aborts. This register is incremented as the depth of nested aborts increases.

The TCB stores the ASR, ATF, and ALC for each thread, and provides this abort context of the current running thread to the AHB. The AHB is a pure combinational circuit.
controlled using a state machine provided to it as an input. The state machine shown
in Fig. 4.3 is actually stored in the TCB instead of the AHB, resulting in a memoryless
design of the AHB. The state machine will be explained later with a larger figure.

When instructed by the main control unit, the AHB checks all abort levels that have
been initialized. If a preemption is taken, an index via CA_SEL that selects the continuation
address stored in AAR. The updated ALC is then sent back to the TCB. The TCB directly
provides the continuation address to the datapath, and hence the AAR is the only part of
the abort context not passed to the AHB. The activation and deactivation of abort levels
are also controlled by the TCB control unit.

The AHB relies on a processor instruction to indicate to it when to check for aborting
conditions. This is necessary to preserve Esterel’s synchronous preemption, and to cor-
rectly implement both strong and weak abortions. This indication from the control unit
is provided using STARPro’s CHKABORT instruction. When the CHKABORT signal arrives,
the AHB control unit will check for aborts in the following manner:

- For strong aborts, the AHB starts by evaluating the status of abort signal beginning
  from the outermost to the innermost abort level. This implementation gives the
  abort constructs priorities determined by the hierarchical order of the aborts.

- Unlike strong aborts, weak aborts allow the abort body to complete the tick before
  terminating it. For this reason, the AHB starts by evaluating the status of aborting
  signals from the innermost to the outermost abort.

As an example of nested strong aborts, the following Esterel program has two levels of
abort. When both S1 and S2 are present, S1 takes high priority and A would be emitted,
but not B. By checking for S1 followed by S2, S1 would be given higher priority.

```
abort
  abort
    halt
  when S2 do emit B end
when S1 do emit A end
```

Nested weak aborts are slightly trickier than strong aborts. In the example below,
when both S1 and S3 are present, but S2 is absent, both A and C would be emitted, but
not B. This behaviour can be implemented by checking from S3 followed by S2 and S1.

```
weak abort
  weak abort
    weak abort
      halt
    when S3 do emit C end
    when S2 do emit B end
when S1 do emit A end
```
A more complicated scenario arises when nested strong and weak aborts are mixed together, such as the example below.

```plaintext
weak abort
   abort
      weak abort
         abort
         halt
         when S4 do emit D end
         when S3 do emit C end
         when S2 do emit B end
         when S1 do emit A end
```

The tricky part of this example is that the weak aborts have higher priority over strong aborts, yet, the strong aborts may still react to the preemption signals. For example, if S1, S3, and S4 are present at the same time, A, C and D are emitted. However, if S1, S2, S3, and S4 are all present, then only A and B are be emitted. This behaviour can be achieved by differentiating the placement of `CHKABORT` instruction of the strong aborts from the weak aborts:

- **strong** aborts are checked at the beginning of each *tick*
- **weak** aborts are checked at the end of each *tick*

This requires the AHB to know the type of abort being checked whenever a `CHKABORT` instruction is executed. The AHB would only check for the type of abort specified by the instruction and ignore the other type. The distinction in behaviours of strong and weak aborts is controlled by the finite state machine (FSM) inside the control unit of the AHB. Based on the type of abort, the condition of a transition triggered between states changes. The FSM depicted in Fig. 4.4(a) contains five states for four levels of aborts, and the transition conditions are summarized in Fig. 4.4(b).

Upon reset of the processor, the FSM is initialized to state A0, where no abort is loaded. The state of AHB is saved in TCB in the ALC register. When the first `ABORT` instruction is executed, the main control unit of the processor activates the LD-AA signal. LD-AA triggers a transition from state A0 to A1 via a01.

The type of abort specified by `CHKABORT` instruction is stored as a single bit in the instruction operand. The operand is passed through the PIR (Pipelined Instruction Register) wire from the datapath. When the `CHKABORT` instruction is executed, the main control unit instructs the AHB to check for preemption. For a strong abort, from a state, for example A4, a transition can be made to any state before it. If all PAEs (Preemptive Abort Events) are present at the same time, PAE1 would be taken, as it is given the highest priority. The transition conditions for the state machine for handling strong aborts
State Description:

A0: Abort empty (no ABORT loaded)
A1: Abort Level 1 (ASR1 and AAR1 loaded with 1st ABORT)
A2: Abort Level 2 (ASR2 and AAR2 loaded with 2nd ABORT)
A3: Abort Level 3 (ASR3 and AAR3 loaded with 3rd ABORT)
A4: Abort Level 4 (ASR4 and AAR4 loaded with 4th ABORT)

State Transition Condition (strong):

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a00</td>
<td>LD AA = '0'</td>
</tr>
<tr>
<td>a01</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a10</td>
<td>(ENDABORT = '1' or PAE1 = '1')</td>
</tr>
<tr>
<td>a11</td>
<td>No operation</td>
</tr>
<tr>
<td>a12</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a20</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a21</td>
<td>(ENDABORT = '1' or PAE2 = '1')</td>
</tr>
<tr>
<td>a22</td>
<td>No operation</td>
</tr>
<tr>
<td>a23</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a30</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a31</td>
<td>PAE2 = '1'</td>
</tr>
<tr>
<td>a32</td>
<td>(ENDABORT = '1' or PAE3 = '1')</td>
</tr>
<tr>
<td>a33</td>
<td>No operation</td>
</tr>
<tr>
<td>a34</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a40</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a41</td>
<td>PAE2 = '1'</td>
</tr>
<tr>
<td>a42</td>
<td>PAE3 = '1'</td>
</tr>
<tr>
<td>a43</td>
<td>(ENDABORT = '1' or PAE4 = '1')</td>
</tr>
<tr>
<td>a44</td>
<td>No operation</td>
</tr>
</tbody>
</table>

State Transition Condition (weak):

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a00</td>
<td>LD AA = '0'</td>
</tr>
<tr>
<td>a01</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a10</td>
<td>(ENDABORT = '1' or PAE1 = '1')</td>
</tr>
<tr>
<td>a11</td>
<td>No operation</td>
</tr>
<tr>
<td>a12</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a20</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a21</td>
<td>(ENDABORT = '1' or PAE2 = '1')</td>
</tr>
<tr>
<td>a22</td>
<td>No operation</td>
</tr>
<tr>
<td>a23</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a30</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a31</td>
<td>PAE2 = '1'</td>
</tr>
<tr>
<td>a32</td>
<td>(ENDABORT = '1' or PAE3 = '1')</td>
</tr>
<tr>
<td>a33</td>
<td>No operation</td>
</tr>
<tr>
<td>a34</td>
<td>LD AA = '1'</td>
</tr>
<tr>
<td>a40</td>
<td>PAE1 = '1'</td>
</tr>
<tr>
<td>a41</td>
<td>PAE2 = '1'</td>
</tr>
<tr>
<td>a42</td>
<td>PAE3 = '1'</td>
</tr>
<tr>
<td>a43</td>
<td>(ENDABORT = '1' or PAE4 = '1')</td>
</tr>
<tr>
<td>a44</td>
<td>No operation</td>
</tr>
</tbody>
</table>

Figure 4.4: Finite State Machine of the Abort Handling Block: (a) The FSM; (b) State transition conditions
are shown in the left column of Fig. 4.4(b). The transition labels are encoded using the labels of the states with the target state the transition leads to. The transition condition is written next to the label. For example, \texttt{a40} would mean a transition from \texttt{A4} to \texttt{A0} triggered by the \texttt{PAE1} signal. A transition from \texttt{A4} to \texttt{A0} would indicate that a preemption had been taken by the outermost abort. If the abort type given by the \texttt{CHKABORT} instruction is weak, the transition conditions on the right column of Fig. 4.4(b) is used. In contrast to strong aborts, a transition from \texttt{A4} to \texttt{A0} via \texttt{a40} is only possible if \texttt{PAE1} is the only \texttt{PAE} signal present. If \texttt{PAE1} and \texttt{PAE4} are both present, then a transition via \texttt{a43} would be taken.

### 4.2 The instruction set architecture

STARPro uses a 32-bit instruction format. Apart from the common instructions found on a typical RISC processor, this section also introduces additional Esterel-oriented instructions to support multithreading, signal testing, and preemption. The syntax and description of these instructions are summarized in Table 4.1.

<table>
<thead>
<tr>
<th>Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SPAWN Reg StartAddr}</td>
<td>Creates a new thread</td>
</tr>
<tr>
<td>\texttt{CSWITCH PriorityVal}</td>
<td>Updates the priority of the current thread (which executes the \texttt{CSWITCH}) to the value of \texttt{PriorityVal}. It then passes control to the scheduler that has to select the next highest priority thread for execution</td>
</tr>
<tr>
<td>\texttt{PAUSE PriorityVal}</td>
<td>In addition to the action of a \texttt{CSWITCH}, it marks the end of a \texttt{local tick}</td>
</tr>
<tr>
<td>\texttt{PCHANGE Reg PriorityVal}</td>
<td>Changes the priority of a thread</td>
</tr>
<tr>
<td>\texttt{PRESENT Sig Reg ElseAddr}</td>
<td>Checks the presence of a signal</td>
</tr>
<tr>
<td>\texttt{ABSENT Sig Reg ElseAddr}</td>
<td>Checks the absence of a signal</td>
</tr>
<tr>
<td>\texttt{ABORT Sig Addr}</td>
<td>Initializes the AHB for strong abortion</td>
</tr>
<tr>
<td>\texttt{WABORT Sig Addr}</td>
<td>Initializes the AHB for weak abortion</td>
</tr>
<tr>
<td>\texttt{WIABORT Sig Addr}</td>
<td>Initializes the AHB for weak immediate abortion</td>
</tr>
<tr>
<td>\texttt{CHKABORT Reg Type}</td>
<td>Checks for preemption of type \texttt{Type} (strong/weak) only</td>
</tr>
<tr>
<td>\texttt{ENDABORT}</td>
<td>Deactivates the current abort level</td>
</tr>
</tbody>
</table>

The number of I/O signal ports is parameterizable. I/O signals are memory-mapped, which enables signal manipulation to be also done using instructions that read from and write to memory. This design allows standard arithmetic or logic operation to be performed on signals. This also provides the flexibility on how signals are interpreted;
this is especially so for valued signals where the value can be represented by a variable number of bits.

Using Fig. 2.1 in Chapter 1 as an example to illustrate how reactive instructions are used, the equivalent STARPro assembly code for that example is shown in Fig. 4.6. For the reader’s convenience, Fig. 2.1 is replicated as Fig. 4.5. The reactive instructions used in this program are explained, while discussion on the translation process is deferred to Chapter 5.

```plaintext
1 module SchizoCyc:
2   input I, R;
3   output C, D;
4   inputoutput B;
5   abort
6   loop
7     signal A in
8       present A then emit B end;
9       pause;
10      present B then emit A end;
11     await A;
12     emit C;
13     | |
14     await immediate I;
15     emit A;
16     present C then emit D end;
17   end signal
18 end loop
19 when R
20 end module
```

Figure 4.5: The SchizoCyc example Esterel program

Starting with ABORT on line 3, one level of abort is configured to watch for signal 14 (signal R). Then, the program forks two concurrent threads. This is accomplished using the SPAWN instruction on lines 8 and 11, which creates two new entries in the thread table of the TCB for thread 1 and 2, respectively. These threads are then initialized to start at labels T1 and T2, respectively. Line 14 creates the special global tick management thread. This special thread is used to interface the reactive program to the environment, and it also preparing the program for the next tick by resetting all the local signals. The PCHANGE instruction on lines 9 and 12 set the initial priority of thread 1 and 2 to 0. Finally, the CSWITCH instruction on line 17 completes the thread-forking process by setting the current (root) thread inactive. The priority number of 255 is the lowest possible priority (indicating an inactive thread), while 0 is the highest. The CSWITCH instruction has two functions – it updates the priority of the thread that executed it, and then, invokes the scheduler. The scheduler in response to CSWITCH, selects a thread for resumption in the
Figure 4.6: The SchizoCyc example translated to STARPro assembly
next instruction cycle. Since both thread 1 and 2 have the same priority, the scheduler can randomly select either thread for execution first. The PAUSE instruction functions similarly to CSWITCH. In addition, it also marks the end of a local tick for the thread that executed it. PAUSE instructions can be found on several lines across thread 1 and 2.

The ABORT instruction initializes one level of abort in the AHB each time it is executed, up to the number of abort levels supported by the AHB. As mentioned earlier, the abort context is stored in the THB. Whenever a PAUSE or CSWITCH is executed, the abort context the AHB receives on its input is also switched to the context of the selected thread. It implies that the abort constructs are kept local to the threads that they have been declared in. This design avoids the need to monitor for preemptions globally using dedicated hardware for each abort. While it simplifies the design of the AHB by monitoring only the current active thread, context switching is now required to check for preemptions in other threads. This severely complicates the scheduling of threads by adding additional context switching requirements to the already complex scheduling problem of Esterel. This complexity can be reduced by replicating aborts for child threads whenever a thread containing aborts is forked, as was done in [11]. The detail of this approach will be explained later in Chapter 5.

The SchizoCyc example is a program that contains two threads inside the body of the abort. The abort is replicated in thread 1 and thread 2 by the compiler. These two child threads begin with an ABORT instruction on lines 29 and 59 respectively. These two lines do the same initialization as was done on line 3. Inside the abort body, the CHKABORT instruction is appropriately inserted at local tick boundaries, such as on lines 36 and 43. As the mnemonic suggests, it checks for the abort at the point of execution of this instruction. It requires a register to be selected and the abort type (strong or weak) to be given. The abortion type operand of a CHKABORT instruction allows the AHB to check only the type of aborts initialized with the same type and ignores the other type. When the end of an abort body is reached, the ENDABORT instruction (see line 49 and 71) is used to deactivate the current abort level, and it will not be checked again until it is reactivated.

The PRESENT instruction, found in many places such as line 32, is functionally equivalent to Esterel’s present statement. If the tested signal is present, the following instruction executes, otherwise the else-address is taken. The ABSENT instruction is similar to PRESENT, except that it checks for a signal’s absence. The STARPro compiler inserts the appropriate conditional branch by anticipating the most probable branch of the condition. For example, abort signals are absent most of the time. If aborts are to be checked in software, the result of the test instructions will most likely be the absent branch. In this case, using ABSENT instructions can improve performance as the number of branching are reduced. Consequently, the frequency of the processor pipeline being flushed is reduced.
Handling aborts in software happens when the number of nested aborts exceed the hardware resource available. For example, if the hardware only supports up to four levels of nested aborts, any additional nested aborts beyond the fourth level will be detected by the compiler and handled in software instead. The compiler inserts \texttt{ABSENT} instructions in place of \texttt{CHKABORT}s. If a \texttt{PRESENT} instruction is used instead, the present branch of the condition would require an additional jump instruction to exit the abort body, and to execute the abort handler. Also, whichever branch of the \texttt{PRESENT} instruction is taken, the processor pipeline will be flushed, which reduces the performance benefit of pipelining.

\section{Execution semantics}

The execution semantics of the Esterel-oriented instructions are mainly driven by the \texttt{CSWITCH}, \texttt{PAUSE}, and \texttt{PCHANGE} instructions. These instructions create, delete, switch between threads, and manage the priorities of the threads. The scheduler keeps the priority of all threads in the TCB. The highest priority is 0, while the number 255 is reserved as an indication that the thread is inactive. The scheduler also stores the \textit{local tick} status flags of every thread. The scheduler makes a decision on which thread to select when a \texttt{CSWITCH} or \texttt{PAUSE} instruction is executed. The decision is made based on the following steps in the given order:

1. Limit the selection to active (priority < 255) threads only.
2. Limit the selection to threads whose \textit{local tick} status evaluates to false.
3. Select the thread with the highest priority (lowest number).
4. When no more threads can be selected, the special \textit{global tick} handler thread is selected and the \textit{local tick} flags of every thread are reset to false. This completes the global tick.

The last step is only performed at the end of a \textit{global tick}, which can only be reached when the \textit{local tick} status flags of all active threads evaluate to true. At the end of each \textit{global tick}, all signal outputs to the environment and local signals need to be reset. A new snapshot of inputs from the environment needs to be taken. STARPro relies on software code to manipulate memory mapped I/O, using a special thread, called the \textit{global tick} handler. The scheduler selects this thread when step 4 above is performed. However, as a single threaded program does not rely on a special \textit{global tick} handler, the code of the \textit{global tick} handler is simply generated in place of the \texttt{pause} nodes for such programs.

To illustrate how SchizoCyc executes on STARPro, the same assembly code is shown in Fig. 4.7(a). On the right, Fig. 4.7(b) shows the changes of the values of the program counter, \textit{local tick} status flag, and the priority of each thread.
Figure 4.7: Change of thread context for SchizoCyc example (a) assembly (b) change of thread context
In the starting instant of the program, the root thread starts forks into two threads. The *global tick* handler thread gets created after line 12, while thread 1 and 2 are created on line 6 and 9 respectively. All threads except the root thread are initialized to a priority of 255 upon start up of the processor. The root thread starts with a priority of 0. During the forking process, the priorities of the threads are re-assigned by the `PCHANGE` instruction, as can be seen in the table at the top of Fig. 4.7(b). We show signals that are present in the current *tick* in the top right hand corner of the thread context table.

Shortly after creating threads, the newly created threads are ready to be scheduled by executing the `CSWITCH` instruction on line 15. The second table below shows the thread context after the fork. The root thread becomes inactive at this point with its priority set to 255. This makes the priority of the parent thread 255 (lowest possible) and passes control to the scheduler. The program counter will be frozen at 15 until the thread is resumed. The same applies to all other threads that are suspended through context switches.

Since both thread 1 and 2 have the same priority, either of them may be selected by the scheduler. If thread 1 is selected, it quickly reaches another context switch on line 25. Looking at the third table from the top, the instruction on line 25 lowered the priority of thread 1 to 1, while thread 2’s priority is now higher at 0. Both thread 1 and 2 have not completed their *local ticks* and are still active, thus both threads still need to be scheduled at some point in the future. Since thread 2 now has higher priority, it is selected for execution first.

The fourth table shows the result after executing thread 2 up to the first `PAUSE` instruction on line 55, assuming input signal I is absent in the first instant. `PAUSE` sets the *local tick* status flag of thread 2 to true, and refreshes its priority with 0. Thread 2 is no longer a candidate for scheduling, leaving thread 1 as the only active thread yet to complete its *local tick*. Similarly, we arrive with the thread context in the next table after completing the remainder of the *tick* in thread 1. No more threads can now be scheduled as thread 0 is inactive, while thread 1 and 2 have both completed their *local ticks*. This triggers a global *tick* signal that is internal to the scheduler, which causes the *global tick* handler thread to be scheduled. Because of the special role of the *global tick* handler, and since it plays no part in the scheduling of threads, it has no priority.

Tick 1 starts after executing the `CSWITCH` instruction on line 23. Note that the *local tick* status flags have been reset to false. The flow of the execution carries on in similar fashion to the previous *tick*.

In a different scenario, an interesting case arises when the input signal I is present in the first instant. Thread 2 would finish before thread 1 in this case. The code between lines 66~73 generated from the *join* node performs barrier synchronization by checking the *JOIN* variable. In this case, as thread 2 finishes before thread 1, thread 2 deactivates
itself without reviving thread 0. Thread 1 continues execution until it also finishes, the same barrier synchronization is performed for thread 1 between lines 45∼52. This time, thread 1 breaks the barrier and revives thread 0 on lines 50∼52.

Lines 16∼18 can only be reached when thread 1 and 2 join. To actually exit the program, thread 0 has to take the preemption by checking for signal R immediately after the join.

4.4 Experimental results

The main focus of the evaluation of the proposed approach is to compare with KEP3a. KEP3a being another architecture-specific multithreaded approach for execution of Esterel, is an ideal candidate to compare to. For completeness, comparison to conventional Esterel compilers were also included. The comparison will be presented in the following order:

1. Hardware resource usage compared to KEP3a.
2. Performance in terms of worst and average-case reaction time.
3. Performance in terms of execution time to complete given input traces compared to conventional Esterel Compilers.
4. Generated code size.
5. Effectiveness of the pipelined architecture.

The first two items will give an insight into the performance of the two architectures given their respective hardware resource requirements. The third item pitches the hardware accelerated execution against non-hardware accelerated execution to gauge the amount of performance gain from hardware acceleration. The forth item shows the difference in generated code size between the architecture-specific and non-architecture-specific approaches. The final item tests how much performance STARPro gains from the pipeline against its non-pipelined version. The first item will now be presented next.

STARPro was successfully synthesized for both Altera Cyclone II [43] and Xilinx Spartan-3 [44] FPGAs. Its hardware resource usage on Spartan3 is presented in Table 4.2 for comparison with KEP3a [12]. STARPro was synthesized for 2 to 512 threads to examine the relationship between the resource usage and the number of threads. Figures for KEP3a have been taken from [12]. These results have been used to produce the graphs in Fig. 4.8.

Both processors exhibit linear increase of required resources (logic gates) with the increase of number of threads. The hardware resource usage grows faster for STARPro
4.4 Experimental results

Table 4.2: Hardware resource usage – STARPro vs KEP3a

<table>
<thead>
<tr>
<th></th>
<th>STARPro @167MHz</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>KEP3a @60MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Threads</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>Gates (k)</td>
<td>24</td>
<td>24</td>
<td>26</td>
<td>29</td>
<td>52</td>
<td>89</td>
<td>173</td>
<td>342</td>
<td>682</td>
<td></td>
</tr>
<tr>
<td>Max Threads</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td>40</td>
<td>60</td>
<td>80</td>
<td>100</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gates (k)</td>
<td>295</td>
<td>299</td>
<td>311</td>
<td>328</td>
<td>346</td>
<td>373</td>
<td>389</td>
<td>406</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

as the number of threads supported increases. However, the hardware resource usage for STARPro is much lower than KEP3a, such that STARPro would only start to require more hardware resources than KEP3a when more than several hundred threads need to be supported. Also worth noting, is that STARPro will not significantly change the hardware resource usage when the number of memory mapped I/O ports increases. KEP3a, in contrast, uses dedicated hardware units to handle signals, which would scale up the hardware resource usage as more signals are being used in the Esterel program.

![Figure 4.8: Comparison of hardware resource usage between KEP3a and STARPro](image)

To evaluate the reaction times of STARPro against KEP3a, several benchmark programs have been selected from EstBench [45], and are listed in Table 4.3. As these programs are the same ones presented in [12], the results can be directly compared. All the Esterel examples used in the benchmark are pure control-driven and have minimal data computation. The compiler optimized results for KEP3a were taken from [12] and reproduced here.

Execution traces were generated using Esterel Studio’s Coverage Analysis tool, as was done in [12]. The Coverage Analysis tool produces a set of input trace that covers all possible states in the program. The worst-case reaction time was obtained from the longest reaction by feeding the generated worst-case input traces to the program. The
 average-case reaction time was obtained by dividing the time required to complete the program with the same worst-case input trace by the total number of ticks required to complete the input trace.

Both the worst-case and average-case reaction times for KEP3a and STARPro are shown in Table 4.4. Although KEP3a has almost one-to-one mapping between Esterel statements and its native instructions, STARPro is still able to achieve, on an average, 37% faster execution (referred to as speedup in Table 4.4) in worst-case reaction time (WCRT), and 38% faster execution in average-case reaction time (ACRT) expressed in number of system clock cycles. The comparison can be considered fair as it assumes that both processors run at the same system clock speed. However, STARPro achieves more than two times higher clock speed than KEP3a (167MHz vs. 60 MHz) when synthesized for the same FPGA device, which gives it even further advantage in terms of real execution time. One exception is that KEP3a significantly excels in performance is the runner example. The example involves counting of signal occurrences. This can be explained by the fact that in KEP3a, such counting is done in hardware, whereas STARPro relies on software to do this.

Table 4.4: The worst and average case reaction time

<table>
<thead>
<tr>
<th>Module Name</th>
<th>KEP3a WCRT (clk cyc)</th>
<th>STARPro WCRT (clk cyc)</th>
<th>Speedup</th>
<th>KEP3a ACRT (clk cyc)</th>
<th>STARPro ACRT (clk cyc)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd</td>
<td>135</td>
<td>83</td>
<td>1.63</td>
<td>84</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>abcdedf</td>
<td>201</td>
<td>121</td>
<td>1.66</td>
<td>117</td>
<td>57</td>
<td>2.05</td>
</tr>
<tr>
<td>eight_but</td>
<td>267</td>
<td>96</td>
<td>2.78</td>
<td>153</td>
<td>87</td>
<td>1.76</td>
</tr>
<tr>
<td>chan_prot</td>
<td>117</td>
<td>140</td>
<td>0.84</td>
<td>54</td>
<td>55</td>
<td>0.98</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>51</td>
<td>43</td>
<td>1.19</td>
<td>39</td>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>runner</td>
<td>30</td>
<td>88</td>
<td>0.34</td>
<td>6</td>
<td>35</td>
<td>0.17</td>
</tr>
<tr>
<td>example</td>
<td>42</td>
<td>46</td>
<td>0.91</td>
<td>24</td>
<td>30</td>
<td>0.8</td>
</tr>
<tr>
<td>Average</td>
<td>137</td>
<td></td>
<td></td>
<td>2.37</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To evaluate STARPro’s compiler against other Esterel compilers, four representative candidates were included. Namely CEC v0.4 [10], EEC2 [29], the V5 [2] and V7 Esterel
compilers [46]. These compilers produced C code from the Esterel source, which we compiled for the NIOS II [47] 32-bit RISC processor. NIOS is a softcore processor, provided by Altera as part of its development tools for its Cyclone II FPGA. Although STARPro has been synthesized on both Altera’s and Xilinx’s FPGA, it was originally implemented using mainly Altera’s technology. The implementation for Xilinx is highly unoptimized, it was done purely for comparing hardware resource usage purposes. For this reason, it would make more sense to execute the C code on an Altera processor. Hence, the NIOS II processor was chosen. All C programs were compiled using the nios2-elf-gcc compiler with level-2 optimization (-O2).

Each Esterel program is executed for one million reactions with randomly generated input traces. Input traces are generated once for each Esterel program. The total number of machine instructions to complete a million reactions is recorded in Table 4.5 and plotted as graphs in Fig. 4.9.

Table 4.5: Performance (in run-time machine instruction count)

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Number of instructions (in millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software approach using C</td>
</tr>
<tr>
<td></td>
<td>CEC</td>
</tr>
<tr>
<td>abcd</td>
<td>347</td>
</tr>
<tr>
<td>abcdedf</td>
<td>473</td>
</tr>
<tr>
<td>eight_but</td>
<td>689</td>
</tr>
<tr>
<td>chan_prot</td>
<td>288</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>97</td>
</tr>
<tr>
<td>runner</td>
<td>181</td>
</tr>
<tr>
<td>example</td>
<td>144</td>
</tr>
</tbody>
</table>

The results have shown that STARPro was able to outperform conventional Esterel compilers by a large margin. The trend of the results are consistent with the trend presented in [12].

To compare the code size, the object files generated by the nios2-elf-gcc compiler were compared for each Esterel compilers. The approach taken by KEP3a and STARPro consistently resulted in much more compact code compared to the conventional software approaches, as depicted in Table 4.6 (plotted as a graph in Fig. 4.10). Overall, STARPro has on an average 40% larger code size than KEP3a. This is expected as Esterel is not completely handled in hardware. It is a trade-off for handling certain features in software, such as valued signals, for flexibility and scalability in terms of hardware resource requirement.

Finally, Table 4.7 shows the performance gain from the pipelined STARPro architecture compared to its non-pipelined version. The clock cycles shown in the table represent the total number of clock cycles to complete each program with a given execution trace. The same applies to the instruction count. Without a pipeline, STARPro requires three
Figure 4.9: Performance (in run-time machine instruction count) comparison in bar graph (the lower the better)

Table 4.6: Code size comparison using different compilation techniques

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Code size (kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CEC</td>
</tr>
<tr>
<td>abcdef</td>
<td>8.94</td>
</tr>
<tr>
<td>eight_but</td>
<td>3.56</td>
</tr>
<tr>
<td>chan_prot</td>
<td>4.41</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>2.25</td>
</tr>
<tr>
<td>runner</td>
<td>4.42</td>
</tr>
<tr>
<td>exaxmple</td>
<td>3.1</td>
</tr>
</tbody>
</table>
In summary, execution of Esterel using reactive processors yields much better code size and execution times compared to conventional software approaches that target traditional processors. The proposed STARPro architecture shows better execution times with significantly less hardware resources compared to the latest KEP processor, but with the larger memory footprint.

In general, the STARPro is simpler than KEP3a in terms of instructions and used functional units. Unlike KEP3a, STARPro does not have a one-to-one mapping of Esterel
statements to its ISA. Instead, it relies on a combination of hardware and software. This approach leads to larger code size compared to KEP3a. However, STARPro has another advantage that it can operate at higher clock frequency when synthesized for the same target FPGA.

4.5 Discussion

This chapter presented a direct execution platform for Esterel with multithreading support. The proposed architecture consists of a specially designed unit to accelerate execution of Esterel attached to the datapath of the processor. Concurrency of Esterel is supported through its Thread Control Block (TCB), thread creation and switching between threads are managed by the TCB. The Abort Handling Block (AHB) is a hardware unit for preemption support. It can be configured to preempt program execution synchronously using the preemption checking instruction at tick boundaries. Optionally, the STARPro compiler supports software preemption when hardware resources are limited.

While the TCB and AHB are hardware units designed to support efficient execution of Esterel, the Esterel semantics is not preserved by the hardware in order to simplify the hardware design. Instead, the semantics are preserved through software. The decision for the hardware and software trade-off made during the design phase has paid off in the experiments. The design greatly simplifies the complexity of the hardware, while also achieving better performance and code size than conventional software approach to compile Esterel.

In comparison to the design of preemption mechanism in KEP3a, the most significant difference between the AHB and the preemption watchers in KEP3a [12] is how the preemption is monitored. STARPro relies on explicit checks at appropriate times using an instruction, whereas the watchers in KEP3a rely on a physical tick signal in hardware. The correctness of abort semantics of the AHB relies on the compiler, whereas the watchers rely on the run-time hardware behaviour. The difference between the two approaches results in simpler preemption hardware design for the STARPro. However, the watcher design of KEP3a scales with the number of nested aborts supported by the hardware. STARPro supports up to 16 nested aborts, it is a limit imposed by the design of the instruction format and the AHB control state machine. The complexity of the state machine of the AHB grows exponentially with respect to the number of nested aborts supported by the hardware. Despite this limitation, the STARPro compiler is able to handle aborts in software in addition to hardware aborts. This has resulted in STARPro achieving better execution times at the cost of larger memory footprint compared to KEP3a. However, the hardware design of STARPro is much simpler, using significantly less hardware resources, provided that the number of threads supported is not excessively high. This design has
also allowed the pipelined STARPro processor to operate at higher frequency at 167MHz. On the other hand, the non-pipelined KEP3a operates at maximum frequency of 60MHz for the same implementation technology.

So far, this chapter has explained the hardware aspect of the architecture specific approach for efficient execution of Esterel. Compilation for such hardware will be explained in the next chapter.
Compiling Esterel for the STARPro Architecture

The previous chapter presented the design of STARPro and the design of the hardware to provide architecture support for efficient execution of Esterel. The process of translating and mapping from Esterel source to STARPro instructions will be explained in this chapter.

The STARPro Compiler first constructs a UCCFG$_{sd}$ as a structured representation of an Esterel program. Following this, threads are scheduled by statically analyzing the data dependencies. The mapping of each Esterel statement to UCCFG$_{sd}$ will be presented first in Section 5.1. Following that, an approach to statically schedule Esterel threads will be explained in Section 5.2.

5.1 The intermediate format

The UCCFG$_{sd}$ is a variant of the UCCFG intermediate format introduced in Chapter 3. However, the UCCFG is not capable of fully preserving Esterel’s semantics, especially for statements that have distinct start and resumption behaviours. These behaviours are known as surface and depth behaviours as introduced in Chapter 3. Some statements, like emit, are logically instantaneous, while others, like await, consumes time (ticks).

To overcome this, the original UCCFG format has been modified, and extended to
explicitly capture both the surface and depth behaviour of every statement in Esterel. This approach adapts the technique used in GRC [21] (also introduced in Chapter 3), where the start and resumption behaviours are differentiated using distinct *surface code* and *depth code*. As an example, the UCCFG$_{sd}$ of the SchizoCyc example is shown in Fig. 5.1.

An UCCFG$_{sd}$ consists of two sub-graphs, a *hierarchical state graph* and a *unrolled concurrent control-flow graph*, or simply control-flow graph for short. The hierarchical graph is the smaller graph on the left in Fig. 5.1, while the control-flow graph is on the right. The hierarchical state graph only represents the skeleton of the program, while the

![Image of a diagram](image)

*Figure 5.1: Unrolled Concurrent Control-Flow Graph of the SchizoCyc example*
5.1 The intermediate format

complete control-flow of the program is captured by the control-flow graph. The nodes in the hierarchical state graph are connected with only parent-child relationships, whereas the nodes in the control-flow graph can have two types of connections. A node in the control-flow graph may have control arcs, data dependency arcs, or both. A control arc is a solid line between two nodes. The control-flow, in general, flows from top to bottom. However, an arrow is used to clarify the direction of the program flow (when needed) such as a jump to a node at the top. A data dependency arc is a dashed line with an arrow indicating the direction of the communication.

For any given node in the UCCFG$_{sd}$ connected to other nodes by a control arc, the nodes that immediately precede the current one are referred as control predecessors, and nodes immediately succeeding the current one are control successors. Similarly, nodes that write data are referred to as data predecessors, while those that read data are referred to as data successors.

For example, the fork node in Fig. 5.1 has two control successors, which are both abort start nodes. The fork node is said to be the control predecessor of the two abort start nodes. An example of a data successor would be the signal test node that tests the output signal $C$. Its data predecessor would be the emit node that writes to the $C$ signal. In the following, the hierarchical state graph will be described first, followed by the unrolled concurrent control-flow graph.

5.1.1 The hierarchical state graph

The hierarchical state graph (HSG) represents the structure of the nested threads and statements. It is mainly used to find positions to insert nodes that check for preemption. As preemption can only happen at tick boundaries, and the state graph preserves the location of each boundary, the exact positions to insert additional nodes can be conveniently tracked from the state graph.

An HSG consists of the following possible nodes, as illustrated in Fig. 5.2:

- **Parallel** node: This node represents a parallel execution state. It can only be extended by thread nodes. When reached, all thread nodes under it are all activated at the same time.

- **Thread** node: This node represents the start of a thread, and a hierarchical state graph always starts with this node. A thread node may precede any of the other types of nodes.

- **Compound** node: This node comes from a compound statement, where one or multiple statements are enclosed inside its defined scope (such as the abort and the signal statements).
• **Boundary node**: This node represents a *pause* statement. As its name suggests, a *boundary* node defines the boundary of a tick.

![Diagram of Hierarchical State Graph Nodes](image)

Figure 5.2: Hierarchical state graph nodes

While the HSG only represents the skeleton of a program, it is the CFG that describes the control-flow of the program in detail. Construction of both the HSG and CFG will be explained in the following sections.

### 5.1.2 The unrolled concurrent control-flow graph

The control-flow graph (CFG) is the part of the UCCFG\_sd that describes the control-flow of Esterel through the primitive nodes illustrated in 5.3. The name *unrolled* comes from the fact that *ticks* are cascaded in sequence, hence, unrolled. This is in contrast to GRC [21] introduced in Chapter 3. Recall that in GRC, each pass of the control-flow graph (CFG) represents an execution of just one *tick*. Thus, to compute the reaction for multiple *ticks*, the CFG would have to be executed within a loop. The selection of the appropriate surface and depth code in each pass of the graph is accomplished using state variables. In contrast, STARPro can directly preserve state information during execution through its PAUSE instruction, which essentially mimics Esterel’s *pause* statement by keeping the program counter for each thread unchanged until the start of the next *tick*. Such encoding is not efficient in general purpose processors, but excels on special hardware like STARPro. Using these *pause* nodes, the loop required to execute the CFG of [21] can be completely unrolled. Hence, instead of using a switch statement to select between the surface and depth code as done in [21], code for STARPro can be conveniently represented in this form:

```
surface(code) followed by depth(code)
```

Each node in the CFG, referred as a control-flow node, is similar to a control flow node in GRC. It describes a simple but distinct operation that can be easily translated to actual code. The control paths of a program is described by the control arcs between nodes, while communication between nodes is described by the data dependency arcs. If
a control arc exists between nodes $n$ and $m$, node $n$ is said to have a control output that goes to $m$, and $m$ has a control input from $n$. Similarly, a node may also have data inputs and outputs. The characteristics of each type of control-flow nodes are briefly described here:

- **Emit node**: This node represents emission of a signal. It has a single control input and a single control output. It also has one or more data outputs that goes to its corresponding test nodes.

- **Test node**: This node represents a conditional statement of a signal expression, such as the presence of a signal. It has a single control input and two control outputs for the then and the else branches of the statement. If it tests on one or more signals that get emitted, it will have one or more data inputs from these corresponding emit nodes.

- **Abort start node**: This node marks the start of the scope of an abort body. The compiler uses a heuristics to generate code that either handles aborts using the AHB, or checks for abort directly in software. The heuristics makes a decision based on the availability of hardware resources to handle aborts. This node will result in actual code being generated if the abort is handled in hardware. Otherwise, nothing will be generated from this node. It has a single control input, a single control output and no data input or output. The heuristics for deciding how this node is translated will be discussed later when the translation of abort is discussed.

- **Check abort node**: This node is a specialized version of a test node. It is only inserted within the scope of an abort body. It has a single control input and two
control outputs. Its then branch (when the preemption needs to be taken) is always connected to the end of the abort body, and the else branch (when preemption is not taken) carries on within the abort body. As a specialized test node, it may also have data inputs from the emit nodes. For example, an abort reacting to a local signal.

- **Abort end** node: This is dual of the abort start node, and marks the end of the scope of an abort body. If the compiler decides to handle this particular abort in software, no actual code will be generated.

- **Fork** node: This node marks the start of a concurrent control flow created by the parallel statements in Esterel. It has a single control input and at least two control outputs. It passes the control to all of its successor nodes.

- **Join** node: This node is dual of a fork node, and marks the end of a concurrent control-flow. It has at least two control inputs and a single control output.

- **Context switch** node: This node has two functions in thread management, which depends on the given parameter: 1) It freezes the current executing thread, then updates the priority of the thread before skipping to another thread. 2) If the priority of the thread is set to the lowest possible value, the thread is killed, and will be removed from the scheduler. The number inside the node is the parameter that represents the priority of the thread to be updated to. This node has a single control input and a control output. However, the control output is not followed immediately, it is only taken when the thread is resumed.

- **Pause** node: This node is similar to a context switch node, but in addition, it marks the end of a local tick of the currently executing thread.

- **Jump** node: This node is an unconditional jump to the target location. It is mainly used to implement conditional statements and loops as STARPro assembly instructions.

- **Assign** node: This node performs an assignment to a variable or a valued signal.

A UCCFG_sd is constructed in three stages: 1) each Esterel statement is visited and translated into a graphical representation using the nodes described above; 2) the check abort nodes are inserted; 3) the data dependencies arcs are inserted between nodes that communicate with each other. Each of the stages will be explained in more detail with an example after the mapping of Esterel statements to UCCFG_sd is described next.
5.1 The intermediate format

The translation process

During the translation process, for each Esterel statement $p$, the compiler repeats the following steps:

1. Create a node $[p]$ as the new root of the HSG, with a sub-graph $SG[p]$ potentially created previously underneath $[p]$. 
2. Translate the surface behaviour by creating the surface graph termed $surface[p]$. 
3. If a depth behaviour exists for $p$, translate the depth behaviour by creating the depth graph termed $depth[p]$. 

When $p$ is composed of more statements, such as a sequence of statements structured as a list of statements, the compiler expands $p$ and applies the steps described above recursively. To make sure that the continuation context of a branching statement is always known beforehand, the compiler traverses the sequential statements in the reverse direction. This special treatment of sequential statements deserves to be elaborated first with an illustration in the next section.

The sequential statements – $p = q; r$

A sequential statement is a container for a sequence of statements. Fig. 5.4 shows the mapping of sequential statements to the HSG and the CFG. The HSG is shown on the left of Fig. 5.4, while the CFG is shown with a surface flow graph and depth flow graph separately on the right. The surface flow graph is always connected to the depth flow graph via the labels shown as $S_n$ or $D_n$. The sequential statement has only one connection from the surface flow graph to the depth flow graph via $D1$.

The statement $p = q; r$ consists of the statement $q$ followed by $r$. The compiler would first construct the sub-graph $depth[r]$, then connect $surface[r]$ to the root node of the $depth[r]$ sub-graph. The same is then repeated for $q$. In the HSG, a thread node is created as the root node $[p]$. Any sub-graph as a result of the statements $q$ and $r$ will be connected underneath $[p]$ as $SG[q]$ and $SG[r]$. 

The compiler traverses the sequential statements in reverse order by starting with $depth[r]$. If for example, $q$ is a present statement with no depth behaviour, the then branch and the else branch can both be joined at the root node of $surface[r]$. If the compiler traverses the sequential statements in forward direction, the compiler would need to keep a history of the previously translated statements in order to connect the sequential statements up for cases like the example described here.
The parallel statements \(- p = q \parallel r \)

A parallel statement is a container for a list of statements running concurrently. Fig. 5.5 depicts the statement \(p = q \parallel r\), the concurrency of the statements \(q\) and \(r\) are represented by extending the fork node in the CFG. A join node is inserted at the end of \(q\) and \(r\) to merge the control-flow. Similar to the CFG, a parallel node is created in the HSG as the root of the thread nodes of \(q\) and \(r\).

The emit statement \(- p = \text{emit} \ S \)

An emit statement is an instantaneous statement and lacks any state information. Hence, nothing is created in the HSG in Fig. 5.6. It is represented by a single emit node in the CFG with no depth behaviour.

The pause statement \(- p = \text{pause} \)

The pause statement is represented as a single node in the CFG in Fig. 5.7. Its shape resembles the context switch node with the addition of a vertical bar to represent a tick boundary. As the pause statement consumes one tick, a boundary node is created to represent this. However, pause does not have any depth behaviour.
5.1 The intermediate format

Figure 5.5: Mapping of $p = q || r$

Figure 5.6: Mapping of $p = \text{emit } S$

Figure 5.7: Mapping of $p = \text{pause}$
The present statement – \( p = \text{present } S \text{ then } q \text{ else } r \text{ end} \)

The present statement is represented using a *test* node to select between the *then* branch \((q)\) and the *else* branch \((r)\) in the CFG in Fig. 5.8. Similar to the sequential statement, the surface graphs will be connected to their respective depth counterparts following the translation of \(q\) and \(r\). A *thread* node is used in the HSG as the root node to hold the sub-graphs of \(q\) and \(r\).

![Diagram of present statement](image)

Figure 5.8: Mapping of \( p = \text{present } S \text{ then } q \text{ else } r \text{ end} \)

The loop statement – \( p = \text{loop } q \text{ end} \)

The translation of the loop statement for the CFG is similar to the sequential statement. In addition, a *jump* node is inserted after the sub-graph \( \text{depth}[q] \), as shown in Fig. 5.9. The *jump* directs the control back to \( \text{surface}[q] \), forming a loop that can only be broken with a preemption or an exception. A *compound* node is created in the HSG to hold the sub-graph \( \text{SG}[q] \).

The signal statement – \( p = \text{signal } S \text{ in } q \text{ end} \)

The *signal* statement is similarly translated as the sequential statement. The difference, which can be seen in Fig. 5.10, is the additional node inserted before the surface graph to
reset the signal. The reset ensures that whenever the scope of the local signal is entered, the signal is initialized to be absent.

**The await statement** — \( p = \text{await } S \text{ do } q \text{ end} \)

The *await* statement can be implemented by adding a loop around a *pause* and a *present* statement. However, this is only true if Esterel has the *goto* feature at the language level. Instead of breaking down *await* by replacing the *await* statement with a combination of other kernel statements, the equivalent behaviour can be achieved easily at the intermediate representation level.

The surface graph in Fig. 5.11 consists of a *test* node, a *jump* node to *surface*\([q]\), and a *pause* node. As the same block of code is shared both in the surface and depth behaviour, the *jump* node here is inserted to avoid duplication of *surface*\([q]\) in both the surface and the depth flow graph. In the depth flow graph, a *jump* node is inserted on the *absent* branch of *test* \( S \) to jump to the *pause* node in the surface flow graph. This is the missing part in Esterel that could have otherwise allowed *await* to be broken down at the language level.

The HSG of the *await* statement is similar to the *present* statement, with the *boundary* node being the only addition.
Figure 5.10: Mapping of \( p = \text{signal} \ S \) in \( q \) end

Figure 5.11: Mapping of \( p = \text{await} \ S \) do \( q \) end
The await immediate statement – \( p = \text{await immediate } S \text{ do } q \text{ end} \)

The immediate version of the \text{await} has a few additional nodes in the surface flow graph. The surface flow graph of Fig. 5.12 consists of a \text{test} node, a \text{jump} node to \text{surface}[q], and a \text{pause} node.

![Surface flow graph and Hierarchical state graph](image)

**Figure 5.12: Mapping of \( p = \text{await immediate } S \text{ do } q \text{ end} \)**

The await statement with multiple cases – \( p = \text{await } S \text{ do } q \text{ case immediate } T \text{ do } r \text{ end} \)

The \text{await} statement may also consist of multiple cases. For each \text{case}, nodes are generated similar to the single case version. These nodes are then cascaded to the absent branch of the previous \text{case}.

Fig. 5.13 shows an example of an \text{await} with two cases. The first case \( S \) is the non-immediate version, while the second case \( T \) is the immediate version. The nodes for the surface behaviour of the immediate case are generated in the surface flow graph, and nodes for both cases are cascaded in the depth flow graph. The order of the cascading follows the order of the cases declared in the \text{await} statement.
The **suspend immediate statement** – \( p = \text{suspend} \ q \ \text{when immediate} \ S \)

The **suspend** statement is handled in two stages. In the first stage, the surface behaviour is translated by inserting a *test* node around a *pause* and a *jump* node. As shown in the surface flow graph of Fig. 5.14, these nodes form a loop such that, as long as signal \( S \) is present, the control stays at the branch with a *pause*. In the second stage, similar loops such as the one in the first stage are inserted at each *tick* boundary inside \( \text{depth}[q] \). These inserted loops will freeze the control whenever \( S \) is present while the suspend is active. The second stage will be explained in more detail after the **abort** statement is discussed. Both of these statements are translated in two stages.

The **suspend statement** – \( p = \text{suspend} \ q \ \text{when} \ S \)

The non-immediate version of **suspend** statement does not react to the signal in the surface behaviour. Suspension can only happen in the depth behaviour. Hence, translating the non-immediate version is simply an expansion of \( q \) in the first stage. Additional nodes are inserted in the second stage to give the suspend statement the *suspend* behaviour.
5.1 The intermediate format

Figure 5.14: Mapping of $p = \text{suspend } q$ when immediate $S$

Figure 5.15: Mapping of $p = \text{suspend } q$ when $S$
The abort statement – \( p = \text{abort } q \text{ when } S \text{ do } r \text{ end} \)

The \textit{abort} statement is another statement requiring a two-stage translation process. To handle the four types of aborts described in Chapter 2, the compiler handles the immediate and non-immediate aborts in the first stage. Then, the behavioural difference between strong and weak aborts is distinguished in the second stage of the translation process. The second stage is also where the nodes are inserted that connect to the abort handler \( r \).

The compiler stores the type of the abort in the \textit{abort start} node in order to distinguish the four abort types. The type of the abort can be visually identified by the letters \( W \) and \( I \) in the \textit{abort start} node in Fig. 5.16. For strong aborts, the abort node will not be labeled, only weak aborts do. The letter \( W \) indicates that the node represents the start of a weak abort, whereas the letter \( I \) indicates the abort is the immediate version.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig516.png}
\caption{Mapping of \( p = \text{abort } q \text{ when } S \text{ do } r \text{ end} \)}
\end{figure}

At the end of the abort body in the depth flow graph, a \textit{jump} node is created to jump over the abort handler \( r \). The first stage of the abort translation is completed by connecting \textit{surface}[q] and \textit{surface}[r] to \textit{depth}[q] and \textit{depth}[r], respectively.

Aborts can be handled in either the hardware or software, the compiler has a command line switch to set the number of nested aborts supported by the hardware. For example, if STARPro has been synthesized to support up to four levels of nested aborts for each
thread, the following program will trigger the compiler to generate code that handles additional nested aborts in software.

```c
abort
  weak abort
  weak abort
  abort
    abort
      p
    when S5
  when S4;
  abort
    q
  when S6
  when S3
  when S2
when S1
```

In the above example, the number of nested aborts exceeds the available hardware resources, the compiler will translate the `abort p when S5 end` using software implementation. The compiler makes a decision when to start using software aborts based on the following:

1. For each thread, the compiler keeps a stack of aborts visited while it visits each statement recursively. It defaults to the hardware abort state. In the above example, the compiler would already have S1~S4 in the stack by the time it reaches S5.

2. When the stack size reaches beyond the number of nested aborts supported by the hardware, the compiler goes into the software abort state. That is, S1~S4 will be handled in hardware, while S5 will be handled in software.

3. When the stack size reduces to less than or equal to the number of nested aborts supported by the hardware, the compiler goes back to the hardware abort state. That is, when the compiler returns from visiting S4, the stack will contain S1~S3. As the stack size is now below the limit of four levels of aborts supported by the hardware, S6 will be handled in hardware when visited.

For aborts that are handled in software, the `abort start` and the `abort end` nodes will not result in actual code being generated. The ability to handle aborts in software and hardware significantly improves the flexibility of the architecture specific compilation approach. The resources available in hardware provide an efficient way to handle aborts, while the hardware acceleration does not impose a limitation on the number of nested aborts the generated code can handle.
The abort immediate statement — \( p = \text{abort } q \text{ when immediate } S \text{ do } r \text{ end } \)

Additional nodes are inserted in the surface flow graph of Fig. 5.17 for the strong-immediate version of the abort statement. The nodes test and jump are inserted before the start of the abort. If the preemption signal \( S \) is present, the abort body \( q \) is skipped over by the jump node, and the abort handler \( r \) is executed.

![Diagram of abort immediate statement](image)

Figure 5.17: Mapping of \( p = \text{abort } q \text{ when immediate } S \text{ do } r \text{ end } \)

The trap statement — \( p = \text{trap } T \text{ in } q \text{ handle } T \text{ do } r \text{ end } \)

Another statement translated in similar fashion to the abort statement is the trap statement, it is also translated in two stages. For example, the following trap statement,

\[
\text{trap } T \text{ in } \\
q \\
\text{handle } T \text{ do } \\
r \\
\text{end}
\]

can be handled as if it were a self-triggered (by the exit statement) weak-immediate abort,

\[
\text{signal } T \text{ in } \\
\text{weak abort}
\]
where the `exit T` statement will be replaced with an `emit T` and a `jump` node to the end of the trap body $q$. If $q$ consists of any parallel statements, the second stage of the translation process will insert checks for the exception at each `tick` boundary.

To sum it up, statements that require a second stage insertion at `tick` boundaries are:

- `suspend`,
- `strong and weak abort`, and
- `trap`.

The second stage implements the strong and weak behaviours of `abort`, and makes sure concurrent threads enclosed within the aforementioned statements react synchronously. The second stage will be discussed in detail next.

### 5.1.3 Handling synchronous preemption

The second stage of the translation process involves placing the corresponding code for checking and reacting to the signals being monitored at `tick` boundaries. The HSG, such as the one for the program in Fig.5.18 can be used to track the `tick` boundaries.

![Diagram](https://example.com/diagram.png)

*Figure 5.18: Tracking the `tick` boundaries using the HSG*

The compiler keeps a stack of `compound` nodes being visited while recursively traversing the HSG. However, only `compound` nodes that correspond to `abort`, `suspend`, and
trap are inserted into the stack. At the top of the HSG, the abort for S1 encloses a pause. The boundary node for this pause is found under the compound node labeled S1. The boundary node contains internally a link to the pause node in the CFG, allowing the compiler to follow the link and insert the check abort nodes. While the compiler visits the abort for S1, it pushes it into the stack. As the abort for S2 is in sequence with S1, it is a separate compound node next to S1. When the compiler visits S2, S1 is pushed out of the stack and S2 is pushed in. However, S3 and S4 are nested inside S2, the compound nodes for them are attached under the S1 compound node one after another, respectively. Each of these compound nodes will be pushed into the stack as the compiler visits them. By the time the compiler reaches the compound node for S5, the stack would already have S2~S4 in it. As the compound node for S5 is a local signal, it is not one of the Esterel compound statements that require node insertion at the second stage. Therefore, that compound node will not be pushed into the stack. When the compiler starts following the three boundary nodes nodes under S5, the compiler is fully aware of the nested aborts that enclosed these nodes.

Nodes for checking aborts, suspension and triggering traps are all inserted this way. The exact nodes inserted varies based on the compound nodes stored in the stack. As abort is the most complex of these statements, it will be explained first.

Depending on the type of the abort, placement of the check abort nodes varies with respect to the tick boundary. To handle the four types of aborts, the following general rules apply:

- A strong abort always checks for preemption at the start of a tick. Therefore, a check abort node is placed immediately after each pause node.
- A weak abort always checks for preemption at the end of a tick. Therefore, a check abort node is placed immediately before each pause node.
- The immediate version of a strong abort checks for preemption before entering the abort body. This would have already been done during the first stage of the translation process.
- The non-immediate version of a weak abort also has the check abort nodes inserted before the pause node of the first instant.

In the following, how the four types of aborts are translated are further elaborated.

Handling the strong abort in hardware

Fig. 5.19 provides an example of a strong abort. The abort body consists of a series of sequential statements, with a pause statement in between each pair of instantaneous statements. The corresponding UCCFG was of the example program is shown in Fig. 5.19(c).
5.1 The intermediate format

Figure 5.19: Mapping of a strong abort: (a) Esterel source; (b) Assembly; (c) UCCFG\text{sd}; (d) Reaction timeline
By applying the translation for the `abort` statement discussed previously, the `abort` and `when S` statements are directly mapped to the `abort start` and `abort end` nodes. Within the abort body after each `pause` node, a `check abort` node is inserted immediately below. This mapping can be compared to the reaction timeline illustrated in Fig. 5.19(d). The timeline illustrates how the program is supposed to react over the `ticks`.

On the first `tick`, the `S` input is present, the program does not have any `check abort` nodes inserted before the first `pause` node. Hence, `X` is correctly emitted and the preemption is not taken. On the second `tick`, `S` is still present, the `check abort` node immediately below the first `pause` node detects the presence of `S`. Subsequently, the preemption is taken this time and `Y` is emitted. The program correctly terminates after the second `tick` completes.

To see how an abort works at the assembly instruction level, and the relationship between the UCCFG_{sd} and low level execution, the generated code is provided in Fig. 5.19(b) for comparison. The assembly version is an exact model of the UCCFG_{sd}. The assembly program defines the start of the abort body by the `ABORT` instruction, and ends the abort body at the label `CONT`. `Pause` nodes are translated to `PAUSE` instructions.

**Handling the strong-immediate abort in hardware**

The immediate version of a strong abort in addition checks for preemption at the beginning of the starting instant (surface behaviour). A `signal test` node has been inserted before entering the abort body (see Fig. 5.20(c)) during the first stage of the translation process, `check abort` nodes are still inserted the same way as the non-immediate version.

Comparing the behaviour to the reaction timeline, the `test` node at the top of the graph detects the presence of `S` immediately in the first `tick`. Subsequently, the program emits `Y` and terminates immediately. Notice that in this case, the `abort start` node is not even executed, the initialization of the abort hardware is completely skipped.

**Handling the weak-immediate abort in hardware**

A weak abort differs from a strong abort with respect to when a preemption is taken. A weak abort allows its body to execute one last time at the instant of preemption. To preserve this behaviour, checking for preemption is done at the end of each tick. Fig. 5.21(c) illustrates how `check abort` nodes are inserted immediately above each `pause` node.

Comparing to the reaction time, the `check abort` node at the end of the first `tick` allows `X` to be emitted prior to taking the preemption. Subsequently, `Y` is correctly emitted in the same tick as `X`.
5.1 The intermediate format

```
1 abort
2   emit X;
3   pause;
4   emit X;
5   pause;
6   emit X
7 when immediate S;
8 emit Y
```

(a)

```
1        ABSENT S0 CONT
2        ABORT S0 CONT
3        LDR RO #1
4        STR RO $X
5        PAUSE #0
6        CHKABORT STRONG
7        STR RO $X
8        PAUSE #0
9        CHKABORT STRONG
10       STR RO $X
11       CONT STR RO $Y
```

(b)

(c)

(d)

Figure 5.20: Mapping of a strong-immediate abort: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline
weak abort
emit X;
pause;
emit X;
pause;
emit X
when immediate $S$;
emit $Y$

WIABORT S0 CONT
LDR R0 #1
STR R0 $?X
CHKABORT WEAK
PAUSE #0
STR R0 $?X
CHKABORT WEAK
PAUSE #0
STR R0 $?X
CONT STR R0 $?Y

Figure 5.21: Mapping of a weak-immediate abort: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline
Handling the weak abort in hardware

The handling of a non-immediate weak abort is subtle when the abort body contains a loop. The first iteration through the loop is different from all subsequent iterations, as the surface part of the loop body gets folded back into the depth after the first iteration. In this case, the abortion condition need not be checked during the first pass of the loop, but would need to be done in subsequent passes. In order to handle this, the Abort Handling Block in STARPro has been designed to ignore the first CHKABORT instruction encountered for weak non-immediate aborts using an additional status bit, referred as the surface flag. The surface flag is only valid for non-immediate weak aborts, and it is initialized to false by the execution of a WABORT instruction, indicating the surface of the body has yet been executed. The surface flag is set on the first CHKABORT instruction in the non-immediate weak abort body, and the hardware skips over this first CHKABORT. The CHKABORT instruction will only work after the surface flag is set for non-immediate weak aborts. This explains why Fig.5.22(b) and Fig.5.22(c) are exactly the same as their immediate counterparts. The abort start node labelled with W is the only clue that the abort is a non-immediate version.

In the example, if the preemption is never taken, the first two ticks will emit X, and the third tick will emit both Y and X due to the loop. To verify that the behaviour is correctly preserved, the mapping will be compared to the reaction timeline. In the first tick, the abort start node initializes the Abort Handling Block with surface flag defaulted to false. When the first check abort node at the end of the first tick is reached, it sets the surface flag without taking the preemption. X is emitted on the first tick, but Y is not as the preemption is taken. On the second tick, S is absent, and X is still the only signal emitted. On the third tick, S becomes present again. The program first executes the emit Y node, followed by taking the jump to the emit X node. With the surface flag set in the first iteration of the loop, the check abort node above the first pause node takes the preemption this time. Then, Z is emitted and the program terminates. X, Y and Z have now been shown to be correctly emitted in the third tick.

Handling abort in software

In the case of software implementation for abort, a test node is inserted in place of check abort nodes. The main difference between the use of a check abort node and a test node is that, a single check abort node can check all of the nested aborts local to the currently executing thread at the same time. A test node, however, can only check one signal at a time. Moreover, multiple test nodes have to be inserted in the same order as strong aborts are nested. The subtle difference between handling aborts in hardware and software is the lack of the abort start and abort end nodes. All four types of aborts are handled
Figure 5.22: Mapping of a weak abort with a loop: (a) Esterel source; (b) Assembly; (c) \text{UCCFG}_sd; (d) Reaction timeline
in similar fashion to the hardware approach. The mapping to the UCCFG$_{sd}$ and the assembly instructions are illustrated in Fig. 5.23, 5.24, 5.25 and 5.26.

```
1 abort
2   emit X;
3   pause;
4   emit X;
5   pause;
6   emit X
7 when S;
8 emit Y
```

(a)

```
1 LDR R0 #1
2 STR R0 $X
3 PAUSE #0
4 ABSENT S0 R1 CONT
5 STR R0 $X
6 PAUSE #0
7 ABSENT S0 R1 CONT
8 STR R0 $X
9 CONT STR R0 $Y
```

(b)

Figure 5.23: Mapping of a strong abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$

Checking for preemption at the assembly instruction level is done using the ABSENT instruction. If the preempting signal is present, the ABSENT instruction jumps to the end of the abort.

In general, handling aborts in software is also very efficient. As long as the nesting of aborts is shallow, handling aborts in software would require approximately the same number of instructions as the hardware version. The only exception is the weak abort. The surface flag has to be implemented in software, requiring a few more instructions than the hardware version.

So far the subtle difference between the mapping of different aborts have been presented. However, note the orthogonality of the four-types of aborts. In particular, the translation for strong-immediate abort is different from that of the weak-immediate ($\text{WI}$). Unlike the strong-immediate abort, which has a present statement guarding the abort to deal with preemption at the starting instant, there is not a similar strategy for the weak-immediate abort. The discussion so far focuses on handling preemptions locally in a single thread. Handling preemption synchronously will be discussed in the following section.
1. `abort`
2.  `emit X;`
3.  `pause;`
4.  `emit X;`
5.  `pause;`
6.  `emit X`
7.  `when immediate S;`
8.  `emit Y`

(a)

```
ABSENT S0 R1 CONT
LDR R0 #1
STR R0 $X
PAUSE #0
ABSENT S0 R1 CONT
STR R0 $X
PAUSE #0
ABSENT S0 R1 CONT
STR R0 $X
CONT STR R0 $Y
```

(b)

Figure 5.24: Mapping of a strong-immediate abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG

1. `weak abort`
2.  `emit X;`
3.  `pause;`
4.  `emit X;`
5.  `pause;`
6.  `emit X`
7.  `when immediate S;`
8.  `emit Y`

(a)

```
LDR R0 #1
STR R0 $X
ABSENT S0 R1 CONT
PAUSE #0
STR R0 $X
ABSENT S0 R1 CONT
PAUSE #0
STR R0 $X
CONT STR R0 $Y
```

(b)

Figure 5.25: Mapping of a weak-immediate abort using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFG
5.1 The intermediate format

weak abort
loop
emit X;
pause;
emit X;
pause;
emit Y
end loop
when S:
emit Z

LDR R1 #0
STR R0 $SURFACE ; surface = 0
LOOP LDR R0 #1
STR R0 $X
LDR R1 $SURFACE
SZ CHK ; if surface == 0
STR R0 $SURFACE ; surface = 1
JMP SKIP
CHK ABSENT S0 R2 CONT
SKIP PAUSE #0
STR R0 $X
ABSENT S0 R2 CONT
PAUSE #0
STR R0 $Y
JMP LOOP
CONT STR R0 $Z

Figure 5.26: Mapping of a weak abort with a loop using software aborts: (a) Esterel source; (b) Assembly; (c) UCCFGsd
Synchronous preemption of concurrent threads

When an abort, suspend or trap body is forked into multiple threads, the abort has to be replicated. This is due to aborts being stored locally to each thread in STARPro, an abort declared in the parent thread cannot be directly checked within its forked body. Instead, each thread within the abort body has to check for preemption individually. Hence, an abort is copied to each child thread by replicating and placing it around threads within its abort body. This approach greatly simplifies the hardware design by relying on the compiler to generate the correct code to check for preemption.

An abort statement with a forked body can be illustrated using the example,

```plaintext
abort
  p
  ||
  q
when S do
  r
end
```

and transform it into,

```plaintext
abort
  abort
    p
  when S
    ||
    abort
      q
  when S
when S do
  r
end
```

The suspend statement

The suspend statement is handled similarly to the abort statement. The difference is, instead of placing check abort nodes, test nodes are inserted, as illustrated in Fig. 5.27. For as long as the signal is present, the program keeps looping around the pause node, in effect freezing the program at that state.

In the first tick of the example, no check is done before the first pause node. This matches the behaviour shown in Fig. 5.27(d). X is emitted despite S is present. In the second tick, the test node immediately below the first pause node detects the presence of S. The control follows the presence branch of the test node back to the first pause node. Subsequently, the control is frozen just before the the first emit X node in the second tick.
5.1 The intermediate format

Figure 5.27: Mapping of a suspend: (a) Esterel source; (b) Assembly; (c) UCCFGsd; (d) Reaction timeline
Then, $S$ is absent in the third tick, the node below the `test` node gets executed and emits $X$.

The immediate version of `suspend` requires `awaiting` on the absence of the suspend signal in the first tick, as illustrated in Fig. 5.28. This is the only difference to the non-immediate version. The effect of the immediate await freezes the program in the first tick until the signal becomes absent. This matches the behaviour in Fig. 5.28(d), where $X$ is not emitted in the first tick. Only when $S$ is absent in the second tick, $X$ is emitted.

1 suspend
2 emit X;
3 pause;
4 emit X;
5 pause;
6 emit X
7 when immediate S;
8 emit Y

(a)

```
1 P1 PRESENT S0 R1 P2
2 PAUSE #0
3 JMP P1
4 P2 LDR R0 #1
5 STR R0 $X
6 P3 PAUSE #0
7 ABSENT S0 R1 P3
8 STR R0 $X
9 P4 PAUSE #0
10 ABSENT S0 R1 P4
11 STR R0 $X
12 STR R0 $Y
```

(b)

```
<table>
<thead>
<tr>
<th>ticks</th>
<th>S</th>
<th>X</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(d)

Figure 5.28: Mapping of an immediate suspend: (a) Esterel source; (b) Assembly; (c) UCCFG$_{sd}$; (d) Reaction timeline

The trap statement

The `trap` statement is more complex to handle than `abort` and `suspend`. Inserting nodes for the `trap` statement can be illustrated using the example in Fig. 5.29.

Different nodes are inserted for checking and throwing an exception using the `exit` statement. In this example, the trap body consists of two threads. The thread on the left
5.1 The intermediate format

```
trap T in
  emit X;
  present S then
    exit T;
  end
  pause;
  emit Y
  ||
  emit Z;
  pause;
  emit X
when S
```

(a)

```
LDR R1 #0
STR R0 $T ; T = 0
; ... start forking
; ----- Start of thread 1 ----- 
T1 STR R0 $X
PRESENT S0 R0 L0
STR R0 $T
JMP L1
L0 PAUSE #0
STR R0 $Y
L1 ; ... start joining
; ----- Start of thread 2 ----- 
T2 STR R0 $Z
LDR R1 $T
ABSENT S0 R1 L2
PAUSE #0
STR R0 $X
L2 ; ... start joining
```

(b)

(c)

```
ticks
X
S
Z
input
output
```

(d)

Figure 5.29: Mapping of a trap in UCCFG_{sd}
side of Fig. 5.29(c) can potentially trigger the exception by executing `exit T`. The `exit` statement is mapped to an `emit T` node followed by a `jump` node. The `jump` node goes straight to the `join` node, forcing the thread to terminate. The thread on the right checks for the exception by testing the signal $T$ just before the `pause` node. If the exception signal is emitted, this thread also terminate synchronously at the `join` node.

5.1.4 Translation example

As an illustration of how an Esterel program is translated from source to the UCCFG$_{sd}$, the example from Section 2.3 will be used:

```e
module SchizoCyc:
  input I, R;
  output C, D;
  inputoutput B;
  abort
  loop
    signal A in
      present A then emit B end;
      pause;
      present B then emit A end;
      await A;
      emit C;
      ||
      await immediate I;
      emit A;
    present C then emit D end;
  end signal
  end loop
  when R
  end module
```

Figure 5.30: The schizocyc example Esterel program

Before applying the mapping for each statement, it would be most convenient to visualize the Esterel source the way the compiler sees it. The Abstract Syntax Tree (AST) is constructed to present the structure of the source. The AST for the example in Fig. 5.30 is shown in Fig. 5.31.

The translation starts from the outer most layer inwards. The first statement encountered is the `abort`. This `abort` is a strong non-immediate type, and does not have an abort handler. Without a surface behaviour and an abort handler, the `abort` is simply mapped to an `abort start` and `abort end` pair in Fig. 5.32(a). In the HSG, due to absence of the abort handler, there is no sub-graph created for the abort handler, leaving just a single branch.
Inside **abort** is a **loop** statement, which in turn encloses a **signal** statement. Both of these statements are represented by **compound** nodes in the HSG in Fig. 5.32(b) and 5.32(c) respectively. Note the **abort end** node in Fig. 5.32(b) becomes unreachable due to the loop. Unreachable nodes are removed during the translation process. The **signal** statement contributes only an **assignment** node in the CFG in Fig. 5.32(c), which is also the start of the loop.

Within the **loop** statement exists a parallel statement. The parallel statement is simply mapped to a fork-join pair in the CFG in 5.32(d). As discussed in Section 5.1.3, any **abort**, **suspend**, and **trap** statement must be replicated whenever the statement body is forked into threads. Here, the **abort** is replicated in the two threads inside its body. Both threads start with an **abort start-abort end** pair, **check abort** nodes will be added after all the statements in the AST have been translated.

After translating the AST in a similar fashion described above, an UCCFGSD is constructed, as shown in Fig. 5.32(e). Upon reaching this stage of the translation process, any **compound** node that does not correspond to an **abort**, **suspend**, or **trap** is removed from the HSG. This compacts the HSG, enabling the syntax-directed insertion of **check abort** nodes in the second stage described in Section 5.1.3.

To insert **check abort** nodes, the compiler traverses the HSG from the top to look for **compound** nodes. The first **compound** node found in the HSG corresponds to the **abort** in the parent thread. It has no **boundary** nodes attached immediately below. When the compiler reaches the **compound** nodes in the child threads, the **boundary** nodes immediately below these **compound** nodes will be used to insert **check abort** nodes. These **boundary** nodes correspond to the two **pause** nodes in the CFG of Fig.5.32(e). Knowing the **abort** is strong type, **check abort** nodes must be inserted below **pause** nodes. Once
Figure 5.32: Translation process of the SchizoCyc example
inserted, the UCCFG$_{sd}$ will look like the one illustrated in Fig. 5.32(f). When the abort condition is true, these check abort nodes lead to the join node, terminating the threads synchronously. A final check abort node is inserted directly below the join to also allow the parent thread to react to the preemption.

The UCCFG$_{sd}$, up to this point, is still yet to be completed. The graph is missing the information about how the program should be executed sequentially. The UCCFG$_{sd}$ will undergo further analysis for thread scheduling before actual code can be generated. This aspect will be discussed in the next section.

5.2 Scheduling

Scheduling of threads is done in hardware at run-time based on the priorities of the threads pre-computed at compile-time. The compiler analyzes the dependencies between threads by traversing the CFG. As described in Section 5.1, nodes in the CFG are connected by control arcs and data dependency arcs. For any given node in the UCCFG$_{sd}$ connected to other nodes by control arcs, these nodes that immediately precede the current one are referred as control predecessors, and nodes immediately succeeding the current one are control successors. Similarly, nodes that write data are referred to as data predecessors, while those that read data are referred to as data successors.

For example, the fork node in Fig. 5.32(f) has two control successors, these are both abort start nodes. The fork node is said to be the control predecessor of the two abort start nodes. An example of a data successor would be the test C node found in the thread on the right branch. Its data predecessor would be the emit C node found in the thread on the left branch.

The next four subsections will present three algorithms that are used to determine how the threads are to be interleaved. These will be explained in the following order:

1. Clustering – this algorithm breaks a program into clusters of control-flow nodes in order to interleave the execution of threads.

2. Priority assignment – this algorithm computes the relative order of clusters such that data dependencies between threads can be satisfied.

3. Inserting context switch nodes – this algorithm inserts context switch nodes when necessary based on the priority assigned to the clusters.

5.2.1 Clustering

The first step is to group the UCCFG$_{sd}$ of a program into clusters of nodes, where each cluster contains the maximum number of control-flow (CF) nodes that can execute before
a context switch is absolutely required. The clustering algorithm is inspired by CEC's [10], and has been modified to work on UCCFG$_{sd}$. This algorithm is presented in Fig. 5.33.

The clustering algorithm uses two sets to store control-flow nodes, a frontier set $F$ that holds nodes that may need to be inserted into new clusters, and a pending set $P$ to hold nodes to be considered for inserting into the cluster being worked on. The algorithm starts by adding the top node of the UCCFG$_{sd}$ to $F$, and arbitrarily selects and moves a node from $F$ into $P$. The outer loop creates a new cluster $C_i$ everytime $P$ becomes empty. The outer loop repeats until all the nodes in the UCCFG$_{sd}$ have been clustered and $F$ becomes empty.

Inside the inner loop, the algorithm first checks if the current node $p$ has been clustered and that $p$ has no data predecessors. However, node $p$ can still be considered for clustering if the current cluster $C_i$ is empty. This is done on line 13 to ensure that an unclustered node with data predecessors can be inserted into a new empty cluster. For example, beginning at the top of Fig. 5.34, the start node is the first to be inserted into the first cluster $C_0$. Its control successor, the abort start node, is then added to $P$ and $F$. A new successor is added to $P$ and $F$ on each iteration of the inner loop until the fork node. The two control successors of the fork node are the abort start nodes. These are inserted into $F$ for clustering in a new cluster later. As soon as $P$ becomes empty, the outer loop restarts, creating a new cluster $C_1$ and adding one of the abort start nodes to it.

Lines 15∼22 in the algorithm specially look for jump nodes. When a jump node leads to a node that has a data predecessor, a context switch node has to be inserted before the jump node. Hence, a jump node should be considered as part of the cluster it is jumping to. The clustering algorithm achieves this by inserting jump nodes into set $F$, then add the successor of the jump node to $P$. This way, the control successor of the jump node is always visited before the jump node. If the successor of the jump node is inserted into a new cluster, the jump node will also be inserted into the new cluster. If $p$ is not a jump node, and it is neither a fork nor a pause, then all of $p$’s control successors are added to the set $P$. By applying the algorithm, the graph is eventually grouped into cluster of nodes, as shown by the grey boxes in Fig. 5.34. Each cluster is bounded by either a pause node or a test node that has at least one data predecessor. A transition from one cluster to another may need a context switch node to be inserted. This depends on whether the succeeding cluster has any data predecessor. The decision is made based on the priority assigned to each cluster by the algorithm to be discussed next.

### 5.2.2 Priority assignment

Following the clustering of all UCCFG$_{sd}$ nodes, priorities are computed by statically analyzing the dependencies between threads. The basic idea of the algorithm is to compute priority values of the clusters such that all signal producers will execute before the con-
C denotes a set of clustered nodes
C\textsubscript{s} denotes the cluster of the control successor of a node
C\textsubscript{i} denotes the current cluster being worked on
i = 0
add topmost CF node to F, the frontier set

\textbf{while } F \neq \emptyset \textbf{ do}
\begin{enumerate}
\item arbitrarily select and remove f from F
\item create a new, empty pending set P
\item add f to P
\item set C\textsubscript{i} to the empty cluster
\item \textbf{while } P \neq \emptyset \textbf{ do}
\begin{enumerate}
\item arbitrarily select and remove p from P
\item if p \notin C \textbf{ and } (p has no data predecessors \textbf{ or } C\textsubscript{i} = \emptyset) \textbf{ then}
\begin{enumerate}
\item add p to C\textsubscript{i} and C
\item if p = jump \textbf{ then}
\begin{enumerate}
\item if p’s successor \in C\textsubscript{s} \textbf{ then}
\begin{enumerate}
\item insert p into C\textsubscript{s} and C
\item insert p into F
\item remove p from C\textsubscript{i} and C
\item add all of p’s control successors to P
\item add all of p’s control successors to F
\item if p \in C \textbf{ then}
\begin{enumerate}
\item remove p from F
\item end
\item if C\textsubscript{i} \neq \emptyset \textbf{ then}
\begin{enumerate}
\item i = i + 1
\item end
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
end
end
Figure 5.33: The clustering algorithm
Figure 5.34: Nodes of the SchizoCyc example grouped into clusters
sumers. This is achieved by determining the longest dependency chain for every cluster. The priority of the cluster at the start of the chain is the highest, while that at the end of the chain is the lowest. All intermediate clusters have incrementally lower priority values. Such an algorithm requires causal programs to compute these chains statically and will not work for non-causal programs. The compiler can, however, generate correct code for programs with certain types of cyclic dependencies.

In Esterel, only non-instantaneous cyclic dependencies are allowed [8]. SchizoCyc is an example of this. It is also possible that an instantaneous cyclic dependency seemingly exists in a program, but does not in reality. Such program would have at least one of the nodes in the dependency cycle that can never be reached in the same instant as the rest. An example of such a program is shown below:

```
1 module InstCyc :
2 input I ;
3 output A, B;
4 present I then
5    present A then emit B end;
6 end
7 ||
8 present I else
9    present B then emit A end;
10 end
11 end module
```

In this example, the first thread reacts to the presence of the input signal I on line 4, while the second thread reacts to the absence of I on line 8. Since I can only be either present or absent at any instant, only one of the statements on lines 5 and 9 can execute. Hence, there are no dependencies between the two threads.

Given a causal program, if a dependency cycle is found, the compiler can still generate correct code. In such a case, the priority assignment assumes the program is causal, and an arbitrary cluster will be chosen as a starting point. The compiler relies on existing tools [13] to do a priori causality analysis prior to compilation. This step is needed to ensure correct code generation using the compiler.

Presented in Fig. 5.35 is the priority assignment algorithm. The first two lines of the priority assignment algorithm do a depth first search along the data dependency arcs of each cluster, where tracing of the data dependency is done by the recursive function `traceDataPred`. Function `traceDataPred` immediately returns the priority of cluster C, the cluster being traced, if the cluster has already been assigned with a priority. This check on line 3 prevents the algorithm from deadlocking in a dependency cycle. An unvisited cluster is initially assigned with a priority of 0. The priority value is updated after all
foreach cluster $C$ in the program do
    traceDataPred($C$)
end

function traceDataPred($C$)
    if $C$ is visited then return priority of $C$
    add $C$ to the visited set
    $max\_depth = 0$
    foreach CF node $n$ in $C$ do
        $depth = 0$
        foreach data predecessor $p$ of $n$ do
            if $n = join$ then
                $n' = $ first non-jump control predecessor of $p$
                $depth = traceDataPred(cluster of n') + 1$
                if $depth > max\_depth$ then
                    $max\_depth = depth$
            end
        else if $p \notin C$ then
            $depth = traceDataPred(cluster of n) + 1$
            if $depth > max\_depth$ then
                $max\_depth = depth$
        end
    end
    assign priority of $C$ with $max\_depth$
    return $max\_depth$
end

Figure 5.35: The priority assignment algorithm
of the data predecessors of the cluster currently visited have been traced. The updated priority comes from the \texttt{max\_depth} variable on line 5 in \texttt{traceDataPred}.

The loop on line 6 traverses through every control-flow node in the cluster and looks for incoming data dependency arcs. The inner loop follows each data dependency arc in a depth first fashion by recursively calling \texttt{traceDataPred}.

A \textit{join} node is specially handled on lines 9\textendash14. This is required because control cannot flow immediately to the join node. Instead, all joining threads need to terminate synchronously before the join node can be reached. To enforce such execution order of the \textit{join} node, the control arcs from the control predecessors of a \textit{join} node are replaced with data dependency arcs. The priority assignment algorithm can then be used to assign the cluster of the \textit{join} node a lower priority than its preceeding clusters.

As mentioned earlier, a \textit{jump} node is grouped into the cluster of its control successor. In the case of a \textit{join} node, it is possible that a \textit{jump} node or a sequence of consecutive \textit{jump} nodes precede the \textit{join} node and reside in the same cluster. Moreover, because the control arcs from these \textit{jump} nodes have been replaced with data dependency arcs, tracing these arcs will only lead to the same cluster. Fig. 5.36 shows an example of such scenario. This creates a problem because the priority cannot be derived by tracing the depth of the dependencies. To assign C4 in Fig. 5.36 a priority, line 10 in the algorithm traverses the UCCFG\textsubscript{sd} upwards until it finds a non-\textit{jump} node \(n'\). Then on line 11, the cluster of \(n'\) is passed to \texttt{traceDataPred} to derive a priority from the preceding clusters of C4. These would be C1 and C2 in Fig. 5.36.

The priority of each \textit{chain} of data dependency arcs are incremented by one, then the maximum priority value of the deepest data dependency chain is assigned to \texttt{max\_depth}.
The intuition is that the higher the value assigned, the lower the priority, and hence, this cluster will execute after all its predecessors. These can be seen on lines 11∼14 and lines 16∼19. The algorithm finishes when all clusters in the UCCFG_{sd} have been visited.

To illustrate how the algorithm works, let us now consider cluster C2 in Fig. 5.34. Assume C0 and C1 have been visited, and C2 is the first cluster that traceDataPred found to have dependencies. C2 has two incoming dependency arcs: one from C8 and another from C4. Assume that the algorithm recursively calls traceDataPred on C4 first. Cluster C4 has only one incoming dependency arc from C2. Thus, creating a cycle.

The arc is traced by traceDataPred(C2). Since C2 is already visited and currently holds a priority of 0, traceDataPred(C2) immediately returns 0. The depth variable in traceDataPred(C4) is assigned with the priority of C2 plus 1. The value of depth, thus, becomes 1. The max_depth variable in C4 then obtains the value 1 from depth. There are now no more dependency arcs to be traced for C4. Hence, traceDataPred(C4) returns 1. At this point, C2 still has one more dependency arc from C8. This is again traced by calling traceDataPred(C8). C8 does not have any dependency, so a priority of 0 is returned. The depth variable in C2 obtains a value of 1. However, this value is less than max_depth. Hence, C2 is now permanently assigned with a priority of 2 (max_depth + 1).

### 5.2.3 Inserting context switch nodes

To interleave between threads, context switch nodes have to be inserted between transitions from one cluster to another. The final algorithm presented in Fig. 5.37 does this. This is done by the context switch insertion algorithm, which examines each cluster in the UCCFG_{sd} by discovering all exit points from the cluster using the loops on lines 1∼2. While traversing the tree, lines 3∼6 of the algorithm uses this opportunity to fill in the priority values in each pause node it encounters. This priority value will become the operand of PAUSE instructions.

The check on line 8 skips over any fork node it encounters. A fork node needs to manipulate the priority of both the thread being forked and its child threads. A CSWITCH instruction will be generated from a fork.

Lines 9∼10 ensure that a minimum number of context switch nodes are inserted. A context switch node is only inserted between a transition from a cluster with a priority value that is lower than its successor cluster. Clusters that depend on each other are maintained in this relative order – signal producers are given a chance to execute before consumer clusters execute. Conversely, a context switch node is not necessary, as the lower priority value of the succeeding cluster means either the dependency has already been satisfied prior to the current executing cluster, or there are no data dependencies between these two clusters. Finally, line 12 stores the priority of the succeeding cluster.
5.3 Handling schizophrenic programs

Statements in an Esterel program may potentially be executed multiple times within a single tick. When a local signal declaration is executed multiple times within a tick, that
Figure 5.38: Completed UCCFG of the SchizoCyc example
5.4 Discussion

Local signal may potentially assume multiple statuses within the *tick*. Such programs are referred to as *schizophrenic* [8, 19]. This phenomenon may result in a single local signal declaration in Esterel being executed multiple times within a *tick*. Esterel compilers typically handle this by creating multiple copies of the same signal (known as *incarnations* [8]) for each new signal declaration that may potentially occur within the *tick*. This not only complicates the compilation process, but also leads to a significant increase in memory footprint due to code duplication.

STARPro’s ISA is able to handle schizophrenic programs correctly without requiring multiple incarnations of a signal. Local signals are simply implemented as variables in STARPro. Whenever the local signal is declared (redeclared on each new iteration of a loop), the corresponding variable will be (re-)initialized. This effectively introduces a fresh copy of the signal by replacing the previous incarnation. This does not pose any problem even for local signals that are shared between multiple threads, as Esterel’s semantics always ensures that parallel statements are synchronously terminated before the local signal enclosing them can be re-declared. This prevents any thread from entering a new scope of the local signal, while other threads are still in the previous scope.

*SchizoCyc* is an example of a schizophrenic program. Emissions of signal $A$ at the end of the threads inside the loop will not be visible in the next iteration of the loop. Signal $A$ will be initialized as absent at the beginning of the loop as can be seen at the top of the control-flow graph in Fig. 5.38. The *join* node acts as a rendezvous point for the two threads, ensuring they will always join before jumping back to the top of the loop to create a new copy of signal $A$.

5.4 Discussion

This chapter has presented an architecture specific approach to compile Esterel for efficient execution on STARPro. The intermediate format used by the compiler, called *unrolled concurrent control-flow graph* with surface and depth ($\text{UCCFG}_{sd}$), is specially designed for natural translation into STARPro assembly instructions. The $\text{UCCFG}_{sd}$ preserves the structure and the semantics of Esterel programs, while it also exposing hardware features, such as the Abort Handling Block, to the compiler for more efficient execution. Compared to the cyclic executive approach taken by [21], $\text{UCCFG}_{sd}$ does not produce duplicated code when dealing with statements such as loops. This ensures the generated code is as compact as possible. Furthermore, the compiler is able to prioritize performance by allocating hardware managed aborts in a greedy manner, but falls back to software implementation when hardware resources are depleted. This hardware-software co-design approach maintains the advantage of performance due to hardware acceleration, and both the flexibility and scalability of software implementation as a fallback solution.
The ability of the compiler to generate code for accelerated abort handling in hardware and in software is one of the highlights of this architecture specific compilation approach. The compiler generates code to handle aborts in software whenever the number of nested aborts exceed the hardware resources available. This flexibility is one of the key factors that differentiates the hardware complexity between STARPro and KEP3a. KEP3a requires an abort watcher hardware unit for each nested abort used in the program, the hardware becomes more expensive as more aborts are required. The design of STARPro is much simpler thanks to its sophisticated compiler. Handling aborts in hardware is not a requirement in STARPro, any number of aborts can be used in the program.
While Esterel provides many features to simplify the specification and verification of embedded systems, it poses several challenges for compilation. In particular, the compiler has to ensure that the logical concurrency of Esterel is suitably “compiled away” so as to preserve the monotonicity of signals (entities using which threads communicate) while allowing “instantaneous dialog” to happen. One aspect that particularly complicates compilation of Esterel is the reaction to absence. These challenges are usually resolved using more recent compilers that are based on a fork-join based approach, linked-list based approach or virtual machine based approach [10]. Similarly, the GRC-intermediate format [21] based compiler, performs several optimizations in the intermediate format to generate very efficient Esterel code. Earlier compilers such as Esterel V3 (automata-based) and V5 (netlist-based) were much less scalable compared to these modern compilers. However, all traditional Esterel compilers were designed for single-core execution of Esterel and will not scale for multicore execution as they lack any approach for suitable thread distribution over multicore platforms. The problems associated with the distribution of synchronous programs is extensively studied in a survey by Girault [48]. Maintaining signal monotonicity, instantaneous dialog and reaction to absence need to be carefully considered to facilitate correct distribution. The problem is even more compounded with multicores as efficiency considerations are paramount.
One of the early works on distribution of Esterel [25] developed a technique over a sensor network and considered correctness while ignoring efficiency. More recent work [49] consider a variant of Esterel (called the Light Esterel Language) that introduces four distinct values for signals (in contrast to only two values for traditional Esterel - a presence or absence) and postpones reaction to absence through a process called finalization. Similar restrictive versions such as synchronous guarded actions [26] generate OpenMP code from synchronous specifications similar to Esterel but do not consider its full complexity. Also, recent work on static timing analysis of synchronous specifications like Esterel over multicores considers a C-based variant (called ForeC [33]), where communication between threads is done using a shared memory combination function that effectively postpones all communication to the next instant. Thus, all these approaches ignore the key challenges posed by multicore execution of Esterel, motivating the current formulation.

To solve these key challenges, we need to consider thread scheduling (which is needed for maintaining signal monotonicity as per Esterel semantics) and thread distribution (which is needed to facilitate good load distribution). We propose to separate these two intertwined problems and make them orthogonal using a new compiler for Esterel. The proposed compiler extends the well known graph code (GRC) [21] intermediate format for Esterel with an approach such that signal statues can be determined at run-time. This facilitates seamless distribution of threads to cores. The proposed signal resolution algorithm is shown to be sound (see Section 6.5 for details).

Two approaches for thread distribution will be presented: an approach that statically distributes threads to cores based on a greedy heuristic, and a dynamic approach where threads are distributed to idle cores at run-time using a FIFO queue. We have implemented the compiler and performed extensive benchmarking using an embedded multicore architecture synthesized using the Xilinx Microblaze processor. Benchmarking is performed over both data-dominated and control dominated programs to reveal that the proposed approach generates efficient code for a large class of Esterel programs. The main contributions of this chapter are:

1. A new compiler for Esterel is proposed that generates efficient code from Esterel programs for multicore architectures. This is the only known compiler for multicore execution of full Esterel V5 language, without relaxing any aspect of its semantics. This is unlike earlier work that either consider a relaxed version of the language or developed new C-based variants.

2. The proposed compiler is founded on a run-time signal resolution approach that is shown to be sound.

3. We have performed extensive benchmarking over an embedded multicore architecture to demonstrate the efficacy of the proposed approach. The developed bench-
marks are based on a standard benchmark suite for Esterel and have been further enhanced with more realistic programs that are amenable to parallel execution.

4. We have also performed benchmarking over an OS-based implementation on a multicore desktop PC to study the performance impact of an OS on the proposed approach. We have shown that data dominated programs are more amenable to parallel execution than control dominated programs.

The remainder of this chapter will proceed as follows: Section 6.1 introduces the run-time signal resolution approach. Then, the implementation of the run-time signal resolution and the intermediate format used are explained in Section 6.2. The static thread distribution approach and the dynamic approach is described in Section 6.3 and 6.4, respectively. Section 6.5 presents the soundness of the run-time signal resolution approach. Evaluation of the effectiveness of these two approaches is presented in Section 6.6 followed by concluding remarks in Section 6.7.

## 6.1 Run-time signal resolution

One of the main challenges of compiling Esterel is the handling of signals. Signals is the only way to communicate with other threads in Esterel. Signals are absent by default and are made present by emissions. Once a signal is emitted, it must retain its presence throughout the tick until its status is reset at the start of the following tick. A signal is deemed absent in a tick only when none of its potential emitters can emit the signal during the current tick. Such semantics guarantees that communication using a signals is always atomic.

To handle the complexity of the signal semantics, Esterel compilers schedule threads statically. The conventional scheduling techniques [20, 21, 10] rely on the analysis of thread dependencies at compile-time to generate sequential code that adheres to the semantics of the Esterel program. However, the reliance on a strict execution order for semantic preservation drastically limits the use of parallelism. In order to maximize the number of threads that can be parallelized, threads should be allowed to execute in any arbitrary order, except during points of communication. This would allow load distribution to be treated as an independent problem. Thus, a key to effective parallelization of Esterel lies in the ability to resolve signal dependencies at run-time, thereby avoiding the sequentialization of threads that arises from compile-time signal resolution. This approach was first developed by Yoong et al. [29].

The run-time signal resolution approach works on the principle that a signal is not read until its status is known. If the status of the signal is unknown when a thread attempts to read a signal, the scheduler freezes the current state of the thread and context switches to
a different thread. Using this approach, threads can be executed in any arbitrary order, while still preserving the communication order imposed by the signal dependencies. One such scheduling strategy could be round robin. A high-level representation of such a cyclic executive is presented in Fig. 6.1. The scheduler itself forms the skeleton of a reactive function. It is invoked once every tick, and returns the termination status of the tick.

The reactive function, on line 5, starts by initializing $S$ with signals that are shared by threads in $T$. The statuses of shared signals are resolved and removed from $S$ as the program progresses over a tick and re-initialized to the unresolved state at the beginning of each tick. Then the root thread of the program is added to a set of threads $T$ as the candidates to be scheduled. The cyclic executive on line 7 is used to continuously schedule threads until $T$ becomes empty. Within the cyclic executive, the first thread $t$ is selected from $T$ for execution. A thread $t$ is a sequential composition of Esterel statements represented as a function $t()$. As threads are scheduled in a round robin fashion, we define one iteration of the scheduler to be completed when every thread in $T$ is scheduled once. If a thread terminates due to completion of a tick, it is removed from $T$ and it will not be rescheduled unless it is added to $T$ again at the start of subsequent ticks. Thus, threads in $T$ are referred as active threads.

We illustrate the run-time signal resolution mechanism through an example in Fig. 6.2. The example in Fig. 6.2 presents a minimal set of Esterel language constructs and introduces bidirectional communication between threads in order to highlight the challenges of scheduling in Esterel. The ResolvingSignals example consists of three threads that communicate with each other through three shared signals ($s_1$, $s_2$ and $s_3$). The table in Fig. 6.2(b) shows an execution trace of the scheduling algorithm in Fig. 6.1 where a tick

```plaintext
1 T denotes a vector of active threads as candidates to be scheduled
2 t denotes a sequential composition of Esterel statements represented as a function
3 S denotes a set of unresolved signals shared by threads in T
4 function reactive_function
5   Populate S with signals shared by $t \in T$
6   add the root thread to T
7   while $T \neq \emptyset$ do // start of the cyclic executive
8       select and remove the first thread $t \in T$
9       $term\_code := t()$
10      if $term\_code = \infty$ then
11         add $t$ to the back of $T$
12      end
13   end
14 return $term\_code$
15 end

Figure 6.1: A cyclic executive for run-time signal resolution
```
6.1 Run-time signal resolution

module ResolvingSignals:
  input I;
  output A, B, C;
  signal s1, s2, s3 in
  % Thread t1
  present s3 then
  emit A
  end;
  emit s1
  ||
  % Thread t2
  present s2 then
  emit B
  else
  emit s3
  end
  ||
  % Thread t3
  present I then
  emit s2
  end;
  present s1 then
  emit C
  end
end signal
end module (a)

<table>
<thead>
<tr>
<th>Iter</th>
<th>Line</th>
<th>Threads ((): line)</th>
<th>Unresolved signals</th>
<th>Resolved signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>{t1, t2 : 12, t3 : 19}</td>
<td>S = {s1, s2, s3}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>{t2, t3 : 19, t1 : 6}</td>
<td>S = {s1, s2, s3}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>{t3, t1 : 6, t2 : 12}</td>
<td>S = {s1, s2, s3}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>{t3, t1 : 6, t2 : 12}</td>
<td>S = {s1, s3}</td>
<td>s2 = 1</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>{t3, t1 : 6, t2 : 12}</td>
<td>S = {s1, s3}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>{t1, t2 : 12, t3 : 22}</td>
<td>S = {s1, s3}</td>
<td>s2 = 1</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>{t2, t3 : 22, t1 : 6}</td>
<td>S = {s1}</td>
<td>s2 = 1, s3 = 0</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>{t3, t1 : 6}</td>
<td>S = {s1}</td>
<td>s2 = 1, s3 = 0</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>{t1, t3 : 22}</td>
<td>S = {}</td>
<td>s1 = 1, s2 = 1, s3 = 0</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>{t3}</td>
<td>S = {}</td>
<td>s1 = 1, s2 = 1, s3 = 0</td>
</tr>
</tbody>
</table>

(b) Figure 6.2: An example of run-time signal resolution
is completed after three iterations (first column) of the cyclic executive. The current execution state is presented in the second column where the current line in Fig. 6.2(a) being executed is shown. The third column shows a set of threads that are yet to complete a tick. The currently executing thread is highlighted in bold in the third column and the resumption context is shown next to each pending thread. The fourth column records a set of currently unresolved signals at each point in the execution trace (initialized by line 5 in Fig. 6.1). The last column lists the statuses of the resolved signals.

The first iteration of the cyclic executive selects and removes $t_1$ from $T$ (line 8 in Fig. 6.1) starts executing $t_1$ (line 9 in Fig. 6.1). On line 6 in Fig. 6.2(a), $t_1$ attempts to read $s_3$ but is prevented from doing so, as the status of $s_3$ is currently unresolved. It immediately returns to the cyclic executive with a termination code of $\infty$. The cyclic executive detects the special termination code and adds $t_1$ back to $T$ (line 11 in Fig. 6.1) with the resumption context set to line 6 of Fig. 6.2(a). The cyclic executive then selects $t_2$ and starts execution from line 12 of Fig. 6.2(a). Similarly, $t_2$ tries to read an unresolved signal $s_2$, it returns immediately and $t_3$ starts to execute from line 19. Note that the status $s_2$ has to be resolved regardless of the status of the input signal $I$. If $I$ was absent, $s_2$ would be resolved as absent. During our illustration, we assume $I$ is present, hence $t_3$ emits $s_2$ on line 20. The number of unresolved signals is now reduced to $S = \{s_1, s_3\}$ since $s_2$ has been resolved due to its emission in $t_3$. Then, $t_3$ continues to execute up to line 22 where it is prevented from reading the unresolved signal $s_1$ and returns to the cyclic executive with a termination code of $\infty$. Since all three threads have now been scheduled, this completes the first iteration of the cyclic executive.

On the second iteration of the cyclic executive, $t_1$ resumes at line 6 but remains blocked by the unresolved signal $s_3$. Thus, $t_1$ immediately returns and $t_2$ resumes at line 12. Since $s_2$ has now been resolved, $t_2$ detects its presence and emits $B$ on line 13. As $t_2$ follows the “then branch” of the present statement, $s_3$ can no longer be emitted. Hence, $s_3$ is now resolved to be absent and only one signal remains unresolved. As $t_2$ has now reached the end, it returns to the cyclic executive with a termination code of zero (signifying completion of a tick by the thread). This time, $t_2$ is not added back to $T$ as the termination code is not $\infty$. Thus, $t_2$ is permanently removed from $T$ until the next tick and the remaining active threads are $T = \{t_1, t_3\}$. The cyclic executive resumes $t_3$, but $t_3$ remains blocked by $s_1$.

On the third iteration, $t_1$ resumes and successfully reads $s_3$. As $s_3$ is absent, $t_3$ does not emit $A$ and continues to emit $s_1$ on line 9. At this point, all shared signals have now been resolved, thus $S = \{\}$ and $T = \{t_3\}$ after $t_1$ terminates. Finally, $t_3$ resumes on line 22, detects the presence of $S_1$ and follows the “then branch” to emit $C$ on line 23.
6.2 Implementation of run-time signal resolution

Implementing the run-time signal resolution algorithm involves two stages of the compilation process. The first stage involves the construction of the intermediate format, and the second involves code generation. We have adopted the GRC intermediate format \cite{21} in the next subsection, as an intermediate format for the proposed approach.

6.2.1 The GRC intermediate format

GRC, introduced in Chapter 3, features a unique way of determining execution path using termination codes from each statement in Esterel. The run-time signal resolution approach elegantly piggy-backs on this special encoding scheme of the termination codes by partially capture scheduling information within the intermediate format. GRC represent synchronous termination of threads by encoding the termination code of each thread in termination nodes at the end of each thread. The termination nodes provide a mechanism for the program to react differently based on the values stored within the termination nodes. The termination mechanism in GRC is augmented with an additional termination code for resolving signals at run-time. Hence, the use of GRC would minimizes the effort required to implement a mechanism for resolving signals at run-time. Compared to the UCCFG_{sd} discussed in Chapter 5, GRC does not require insertion of control logic into the graph for run-time signal resolution.

To support resolving signals at run-time, the GRC format has been augmented with two additional nodes. We will illustrate the two additional nodes using the example introduced in Chapter 3. The ParallelData example is reproduced in Fig. 6.3, while its corresponding representation in GRC augmented with run-time signal resolution nodes is presented in Fig.6.4.

The program starts by testing \( S_0 \), which is initially set to 2, using the switch node at the top of the graph. The switch node determines the current state of the program. Here, branch 2 of \( S_0 \) represents the initial state of the program. A guard (\( S_2? \)) node is inserted in places where a signal must be protected from being read prematurely. A resolution (\( S_2! \)) node is inserted at suitable points to remove a thread from the set of potential emitters for \( S_2 \). For this example, a guard node has been inserted immediately before the test node of \texttt{selfDiagnose(?S2)}, while resolution nodes have been inserted immediately after the emit (\( S_2 \) = final) node, and on all paths that would not lead to \( S \) being emitted. The resolution nodes inserted immediately below signal emissions immediately declares the signal as present. However, other resolution nodes remove the thread from the set of potential emitters.

Synchronizing threads to tick boundaries is achieved explicitly in GRC as part of the control-flow by forking and joining threads at the start and the end of of each tick
module ParallelData:
  input Start, CheckStatus, Stop;
  output Good, Error;
  type Data;
  procedure processData1(Data)();
  procedure processData2(Data)(Data);
  function selfDiagnose(Data): boolean;
  abort
  signal S : Data, S2 : Data in
  var result : Data, final : Data in
    await Start;
    loop
      call processData1(result)();
      emit S(result);
      pause;
    end loop | |
    loop
      await pre(S);
      call processData2(final)(pre(?S));
      emit S2(final);
    end loop |
    every CheckStatus do
      if selfDiagnose(?S2) then
        emit Good;
      else
        emit Error;
      end if
    end every
  end var
  end signal
  when Stop
  end module

Figure 6.3: The ParallelData example written in Esterel
Figure 6.4: The ParallelData example represented in the Graph Code Format

respectively. This behaviour is exemplified by the fork-join pair in each tick when concurrent statements execute in Fig. 6.4. In the ParallelData example, both the initial tick (the left branch under the top switch node) and the subsequent ticks (the middle branch under the top switch node) start by forking into two threads and joining them at the bottom of the GRC graph. This explicit tick synchronization mechanism at the intermediate representation saves the cyclic executive in Fig. 6.1 from having to handle tick synchronization.

Implementing run-time signal resolution at the GRC representation level requires insertion of guard nodes and resolution nodes at appropriate points. While inserting guard nodes is simple, inserting resolution nodes is more involved. A guard node is inserted before a test node whenever the test node reads a signal that is potentially emitted from at least one other thread. In contrast, inserting resolution nodes involves an algorithm that will be described in the next subsection.

6.2.2 The insertion of signal resolution nodes

During the compilation process, the compiler inserts a guard node before a test node to prevent the signal from being tested prematurely, i.e., when the signal status is yet to be resolved. Execution control gets past a guard node only when the status of the signal is resolved. It takes only one potential emitter that emits the signal in order to confirm the presence of the signal. However, determining absence of a signal requires confirmation from all potential emitters. To further complicate the matter, threads often communicate back and forth. Such threads cannot wait till the tick boundary to determine status of
the shared signals if those signals are guarded. Those threads would deadlock due to indefinitely waiting for each other to resolve the statuses of the signals. For this reason, the compiler must insert resolution nodes for each potential emitter to ensure they keep the signal readers informed about the status of the shared signal.

To insert resolution nodes, the insertion algorithm has to find the closest common parent node. A common parent is a conditional node under which the signal producer and consumer are attached. The highlighted switch node at the top of Fig. 6.5 is an example of a common parent of the emit S (i.e., S = 1) node and the test node for S. The highlighted node at the bottom is the signal producer, and the highlighted node in the middle is the consumer. The producer and the consumer execute concurrently under the common parent node as they are in two different branches of the fork node. Hence, a dependency exists in the example in Fig. 6.5 that needs to have guarded access to the shared signal.

![Diagram of places to insert resolution nodes](image)

Figure 6.5: Illustration of places to insert resolution nodes

Finding the common parent node is only the first step to insert resolution nodes. Not all paths from the common parent node would lead to the emit node. As soon as the potential emitter deviates from the paths to the emit node, it must go through a resolution node. For example, the paths to the emit node are highlighted in red in Fig. 6.5. From the common parent onward, any path deviating from the path in red needs to have a resolution node inserted. The intuition is to find all the conditional nodes on the path from the emit node to the common parent node, and then insert resolution nodes under the branches of those conditional nodes except the branch that leads to the emit node. This is the basis for the insertion algorithm in Fig. 6.6.

This algorithm starts with the invocation of insert_resolution_node in Fig. 6.6.
Let $C$ denote a set of nodes in a causal program.

**procedure** insert_resolution_nodes

```plaintext
foreach node $t \in C$ do
    if $t$ is a test node then
        foreach data predecessor $e$ of $t$ do
            insert_nodes($t,e$)
        end
    end
end
```

**procedure** insert_nodes($t,e$)

```plaintext
insert a resolution node under $e$
$P := \text{path_to_node}(e)$
$Q := \text{path_to_node}(t)$
$c := \text{common_parent_node}(P,Q)$
$n := e$
foreach conditional node $p \in P$ do
    if $p = c$ then return
mark_path_to_node($n,p,e$)
$n := p$
foreach branch $b$ under $p$ do
    if $b$ is not on the path to $e$ and a resolution node $r$ for $e$ has not been inserted under $b$ then
        insert $r$ at the top of $b$
end
end
```

Figure 6.6: Algorithm for inserting signal resolution nodes
The auxiliary functions used by \texttt{insert\_nodes} are presented separately in Fig. 6.7. The

\begin{verbatim}
1 \ P\ denotes a vector of conditional nodes on the path to a node from the root node
2 function \ path\_to\_node(e)
3 \quad n := e
4 \quad \textbf{while} control predecessors of \ n > 0 \ \textbf{do}
5 \quad \quad n := the first control predecessor of \ n
6 \quad \quad \textbf{if} \ n \ \textbf{is not a} \ \textbf{fork node} \ \textbf{then}
7 \quad \quad \quad \textbf{add} \ n \ \textbf{to} \ \ P
8 \quad \quad \textbf{end}
9 \quad \textbf{end}
10 \quad \textbf{return} \ P
11 \end
12 \end
1 \ function \ common\_parent\_node(P,Q)
2 \quad \textbf{foreach} \ p \ \in \ P \ \textbf{do}
3 \quad \quad \textbf{foreach} \ q \ \in \ Q \ \textbf{do}
4 \quad \quad \quad \textbf{if} \ p \ = \ q \ \textbf{then} \ \textbf{return} \ p
5 \quad \quad \textbf{end}
6 \quad \textbf{end}
7 \end
1 \ function \ mark\_path\_to\_node(n,c,e)
2 \quad \textbf{if} \ n \ = \ c \ \textbf{then} \ \textbf{return} \ \textbf{true}
3 \quad \textbf{foreach} \ \textbf{control predecessor} \ p \ \textbf{of} \ n \ \textbf{do}
4 \quad \quad \textbf{if} \ mark\_path\_to\_node(p,c,e) \ \textbf{then}
5 \quad \quad \quad \text{mark the branch under} \ c \ \textbf{that} \ n \ \textbf{belongs to as a path to} \ e
6 \quad \quad \textbf{end}
7 \quad \textbf{end}
8 \quad \textbf{return} \ \textbf{false}
9 \end
\end
\end{verbatim}

Figure 6.7: Auxiliary functions for the insertion algorithm

algorithm is initiated with the \texttt{insert\_resolution\_nodes} procedure. On line 3, the
algorithm traverses through the GRC of a program searching for \textit{test} nodes. On line 5,
the procedure follows each \textit{data predecessor} \textit{e} of the \textit{test} node \textit{t} and passes both \textit{t} and \textit{e} to the \texttt{insert\_nodes} procedure. The term \textit{data predecessor} refers to those nodes that
have data dependency arcs leading to other nodes.

Within \texttt{insert\_nodes}, it immediately inserts a \textit{resolution} node under the \textit{e}. The
\textit{resolution} (i.e., \textit{S!} = 0) node inserted under \textit{e} propagates the presence of the signal,
allowing the signal to be consumed. An example of this node is the node labeled with
\textit{S!} attached immediately below the highlighted \textit{emit} node in Fig. 6.5. The \texttt{path\_to\_node}
function on Lines 3 and 4 create two vectors of nodes on the paths to the \textit{emit} node
(the \textit{P} vector) and the \textit{test} node (the \textit{Q} vector) respectively. The \texttt{common\_parent\_node}
function on line 5 finds the closest common parent node of the emit node and the test node by comparing P and Q. Once the common parent node is found, common_parent_node returns to the insert_nodes procedure. Line 6 and the loop on line 7 start searching from the emit node towards the common parent node to find additional places to insert resolution (i.e., S!) nodes. Line 9 checks whether the common parent node has been reached. The procedure then returns to insert_resolution_nodes and move on to the next data predecessor; otherwise, the search continues. The mark_path_to_node function on line 10, presented in Fig. 6.7, recursively traverses each path of the given node ‘n’ upward. It stops traversing as soon as the target node c is reached, and marks the branch under c that n belongs to as a path that will eventually reach e. The insert_nodes procedure calls mark_path_to_node on a segment of the path between the common parent node and the emit node at a time, starting from the emit node to the nearest conditional node in vector P. The segment being searched is changed in line 11 in Fig. 6.6. The segments between c and e that are marked by mark_path_to_node are illustrated with the paths highlighted in red in Fig. 6.5. Lines 12 ∼16 then insert resolution nodes under each conditional node for each branch that deviates from the path to the emit node.

Running through the insertion algorithm on the example in Fig. 6.5 would result in resolution node being added under each branch of each conditional node on the path between the common parent node and the emit node. The algorithm would correctly identify the path marked in red, and avoid inserting resolution nodes on this path. However, the insertion algorithm assumes that all dependencies are valid. It would not work if the dependency is false. An example of both a valid and a false dependency is illustrated in Fig. 6.8.

![Figure 6.8: An example of (a) a valid and (b) false dependency](image)

Fig. 6.8(b) shows an example of false dependency. In this example, the emit and the test nodes are located in different ticks as they are located on different branches of the switch node. The signal is emitted and tested in different ticks. Due to the compiler plainly inserting dependencies between nodes by tracing the data successors or predecessors of a node, some of the matched dependencies are not real dependencies. This will create problems for run-time signal resolution in two ways: (1) it creates extra schedul-
Compiling Esterel for Multi-core Execution

...ing overhead, or worse, (2) potentially produce incorrect behaviours due to attempts to resolve signals due to false dependencies. Hence, the algorithm in Fig. 6.9 is applied prior to the insertion algorithm to remove any false dependencies.

1 \( C \) denotes a set of nodes in a causal program
2 procedure remove_false_dependencies
3     foreach emit node \( e \in C \) do
4         if data successors of \( e > 0 \) then
5             foreach data successor \( t \) of \( e \) do
6                 \( N := \text{path_to_node}(e) \)
7                 \( M := \text{path_to_node}(t) \)
8                 \( c := \text{common_parent_node}(N,M) \)
9                 \( p := \text{the branch under } c \text{ that leads to } e \)
10                \( q := \text{the branch under } c \text{ that leads to } t \)
11                if \( e \) and \( t \) are in the same thread or \( p \neq q \) then
12                    break the dependency between \( e \) and \( t \)
13             end
14         end
15     end
16 end

Figure 6.9: Algorithm for removing false dependencies

Determining whether a dependency is valid is very simple. The algorithm would only need to find the closest common parent of the emit-test node pair that are linked by a data dependency arc. Then, it determines whether the paths to each of the nodes of the pair reside on the same branch under the common parent node. If the pair reside on the same branch like the highlighted branch in the example in Fig. 6.8(a), the dependency is valid. If the dependency is invalid, the pair of nodes would reside on different branches of the common parent node, as illustrated in Fig. 6.8(b). This is precisely what the algorithm in Fig. 6.9 does.

The algorithm for removing false dependencies starts by looking for emit nodes that have at least one data successor. It then finds the closest common parent node on lines 6~8. The branches that lead to the pair of nodes are compared to determine whether they reside on the same branch on line 11. In addition to that condition, line 11 also checks if the pair of nodes reside in the same thread. If either of these conditions is true, the dependency between the nodes would be removed.

6.2.3 Generating code with run-time signal resolution

The second stage of the implementation for run-time signal resolution takes place during code generation. We will first describe the implementation for sequential execution before
we extend this scheduling scheme for parallel execution in Section 6.3 and 6.4.

To avoid complicating the discussion of the implementation with the full details of an actual generated code, we will illustrate a sketch of an Esterel program using the example in Fig. 6.10. The sketch example consists of four threads communicating via the local signal S. The thread at the top is the only reader of S while the other three are potential emitters to S. The GRC representation of this example is presented in Fig. 6.11 to reveal a little more detail of each thread. The GRC representation consists of a fork node spawning four threads. The left three branches of the fork node in Fig. 6.11 lead to the potential emitters of S while the consumer of S is located on the right most branch of the fork node. The test node in each potential emitter represents some condition that decides whether S is emitted. A resolution node has been inserted under each emit S node to inform the consumer about the presence of S. An additional resolution node has been inserted on the non-emitting side of each of the test nodes inside the potential emitters to S. These resolution nodes have completely covered all paths within their respective threads such that these threads would not be able to terminate without first hitting a resolution node. To protect against premature access to S, a guard has been inserted above the ‘test S’ node. Assuming the compiler unwisely schedules the consumer thread before the potential emitters, the signal would be unresolved when this thread reaches the guard node. It then takes the right branch of the guard node and reaches a termination node with a value of ∞. This special termination code falls through the join node when threads join at the join node and the program exits the graph with a value of ∞. Due to this, the cyclic executive detects the special termination code and reschedules the consumer thread in the next iteration.
When the compiler generates code from the GRC representation, the guard nodes are translated into counting semaphores around the shared signals. Each unique shared signal in the program has an associated counting semaphore. The counting semaphores are initialized to the number of potential emitters of their corresponding signal at the beginning of each tick. The counting semaphores are decremented by the resolution nodes in the potential emitters. The guard nodes are used for blocking the program when it attempts to read a signal with a non-zero counting semaphore. This locking and unlocking mechanism is exemplified by the generated code of the sketch example in Fig. 6.12.

The generated code implements the sketch Esterel module in a reactive function called esterel_module. The reactive function is called once every tick. The generated code presents the four threads of the sketch example as four functions. The actual implementation of the compiler would inline these functions into the reactive function. However, threads are shown as functions to aid the presentation of the program structure in the generated code.

The first thing the reactive function does is to initialize the program counters of the four threads. The program counters are used for resuming the threads when they are blocked by a signal guard. Following that, the signal guard is initialized by assigning the lock lock S with the number of potential emitters on line 3. The name of the lock suggests that it is used for protecting the signal S. The cyclic executive is implemented by the loop on lines 5∼11. On every iteration, the loop condition checks the termination code. The loop will continue as long as the value of the termination code equals infinity. Within the loop body, threads execute one after the other in a sequential order. The returned termination code from each thread is combined at the end of each iteration. The
6.2 Implementation of run-time signal resolution

```c
int esterel_module() {
    int thread_1_pc = thread_2_pc = thread_3_pc = thread_4_pc = 0;
    lock_S = 3; // 3 potential emitters
    // The fork node spawns four threads
    do {
        term_1 = thread_1();
        term_2 = thread_2();
        term_3 = thread_3();
        term_4 = thread_4();
        term_0 = term_1 | term_2 | term_3 | term_4;
    } while (term_0 == INFINITY);

    int thread_1() {
        if (thread_1_pc)
            goto *thread_1_pc;
        // ...
        RESUME:
        if (lock_S) {
            thread_1_pc = &&RESUME;
            return INFINITY;
        } else {
            // Inside guarded body
        }
        return 0;
    }

    int thread_2() {
        if (thread_2_pc)
            goto *thread_2_pc;
        if (signal.I) {
            signal.S = 1;
            lock_S = 0;
        } else
            lock_S--;
        END:
        thread_2_pc = &&END;
        return 0;
    }

    int thread_3() {
        if (thread_3_pc)
            goto *thread_3_pc;
        // ...
        lock_S--;
        // ...
        END:
        thread_3_pc = &&END;
        return 0;
    }

    int thread_4() {
        if (thread_4_pc)
            goto *thread_4_pc;
        // ...
        lock_S--;
        // ...
        END:
        thread_4_pc = &&END;
        return 0;
    }
}
```

Figure 6.12: The generated code of the sketch example with run-time signal resolution
combination of termination codes implements the join node in Fig. 6.11, and a value of infinity from a thread would always override the termination code from other threads. The combined value is then checked by the loop condition on line 11 to reschedule the blocked threads. The implementation of the cyclic executive bears some resemblance of its abstract form in Fig. 6.1 of Section 6.1. The difference is that the abstract algorithm removes a thread from the vector when a thread dies; whereas the implementation of the algorithm in Fig. 6.12 controls the threads via their program counters. In the implementation, a thread would not execute if its program counter is zero and it would immediately return to the cyclic executive in esterel_module.

To illustrate how signal locking and unlocking is performed, consider the consumer thread that has been deliberately scheduled before the potential emitters. The actual implementation of the compiler attempts with its best effort to schedule potential emitters before their consumers. In Fig. 6.12, when thread_1 executes, lock_S has still not been released. This causes thread_1 to store the program counter for resumption in the next iteration, and returns a value of infinity (lines 18∼20). Returning to esterel_module, the next thread thread_2 gets scheduled. Depending on the status of the input signal I, S may or may not be emitted. If S is emitted, the code generated from the resolution node unlocks the guard node by setting the counting semaphore to zero. As soon as the signal guard is released, the dependency of the first thread is resolved. The first thread can then be freely rescheduled at any time within the tick. However, if the input signal I is absent, line 33 which corresponds to the resolution node on the else branch is executed and decrements the counting semaphore. With one potential emitter terminated and two remaining, the value of the counting semaphore drops to two and the signal guard remains locked. Assuming both thread_3 and thread_4 do not emit S, the counting semaphore is decremented twice by these two threads and becomes zero. When the thread_1 gets rescheduled on the second iteration of the cyclic executive, it resumes at the address pointed by its previous program counter. Then, it successfully get through the signal guard this time. Since the cyclic executive merely iterates through a static list of threads, threads that have already completed their local ticks are not removed from the list. For this reason, program counters are used to keep those threads at the completion state. Whenever a completed thread is rescheduled, it immediately returns.

The previous sections and this section have discussed the dynamic scheduling approach using run-time signal resolution and the associated implementation. While the run-time signal resolution approach allows Esterel threads to execute in any order for any number of times, the macro behaviour of the program remain the same as a statically scheduled Esterel program. This flexibility elegantly decouples scheduling from thread distribution for executing Esterel programs on a multicore. Thus, thread distribution can be solved as an independent research topic. The next two sections propose a static and a dynamic
approach to distribute Esterel programs.

6.3 Static load distribution

The static distribution approach is an attempt in this research to study the feasibility of executing Esterel on a multicore. Threads are statically grouped together based on the amount of computation within each thread approximated by a heuristic guided algorithm. Each group is distributed to a dedicated core and no thread will be migrated between cores at run-time. We will describe the partitioning algorithm in the next subsection, followed by the implementation for the partitioned programs in section 6.3.2.

6.3.1 The static load distribution algorithm

The process of grouping threads together, called partitioning, takes place after the GRC of the program has been constructed. The GRC representation of the program is analyzed by a distribution heuristic to partition the nodes under the parallel branches of a fork node. When the compiler generates the code from the GRC, each partition is enclosed in a cyclic executive such that threads within a partition are scheduled independently of other partitions. As each processor executes its assigned partition in parallel, they all synchronize at each tick boundary. However, the barrier synchronization of the partitions at tick boundaries imposes some rules over how threads should be partitioned:

The atomicity of threads: A thread should not be divided to execute in multiple partitions. The whole thread should be allocated to one partition. Failing to follow this rule would incur performance penalties due to resolving control dependencies in addition to data dependencies.

The hierarchic ordering of threads: If a program (p||q) is distributed to partitions P1 and P2 respectively, any child thread forking from p should not be allocated to P2, and vice versa for q. Failing to follow this rule would incur performance penalties due to synchronization for starting forks in addition to synchronous joins.

Based on these rules, the distribution algorithm only distributes the threads spawned by the top level fork nodes in the root thread. An example of a partitioned program is illustrated in Fig. 6.13. The GRC representation shows the ParallelData example partitioned to execute on two CPUs. When the program starts, the root thread executes on the first CPU. As soon as the program reaches a fork node, the nodes shaded in gray are spawned on the second CPU, while the non-shaded nodes are spawned on the first CPU.
Based on the amount of code the compiler would generate for each type of node in the GRC, a relative cost is assigned to each type of node. The cost of a node is approximated by the number of instructions required for that node and normalized against the node with the least cost. For example, an emit node would in general have the lowest cost. It requires only an assignment operation in the generated code. Hence, an emit node would have a cost of one unit. For a fork node, an operation is required to initialize each thread it spawns. The cost of a fork is then approximately the number of threads it spawns. For example, to spawn four threads, the cost of the fork node would be four units. That is four times the cost of an emit node. The cost of a host procedure call is difficult to determine without exhaustive timing analysis. For nodes that call host procedures, the compiler makes an approximation of the cost by summing up the number of instructions of the host procedures. Other types of nodes have the same cost of one unit. Then, the partitioning algorithm computes the total cost of each thread by summing up the cost of each node within each thread under the top level fork node. Only top level fork nodes in the root thread are partitioned due to the rules described earlier.

After the cost of each node (represented by c in the algorithm presented in Fig. 6.14) in the GRC has been approximated, the partition_dfs algorithm in Fig. 6.14 starts partitioning using a depth-first search through the GRC. The algorithm distinguishes only fork and join nodes from the other types of nodes. More specifically, the algorithm looks for the top level fork and join nodes in the root thread. Other types of nodes are treated as ordinary nodes and their cost is added to the total cost of the thread being
traversed. The identification of the top level fork is done on line 11. If a top level fork node, $r$, is found, lines 12 and 18 keep track of $r$ until the costs of all branches under $r$ have been estimated. Any fork node nested under $r$ will cause the condition on line 11 to fail and the nested fork nodes are treated as ordinary nodes.

When a top level fork node is found, the loop on line 13 starts traversing the threads under $r$ on a branch-by-branch (thread-by-thread) basis and assigns the reference of the root fork node to $r$. In the $r \neq 0$ state, the algorithm recursively traverses down each branch (thread) under $r$ until it is stopped by the condition on line 9 when the corresponding join node of $r$ has been reached. Before traversing each thread under $r$, line 14 initializes a new cost $c$ with zero for the thread about to be visited and adds $c$ to $C$. Then, the algorithm recursively calls partition_dfs on the first node of each thread. For each node that partition_dfs visits, the ordinary nodes would fail both conditions on line 9 and 11 and start from line 24. As $C$ is not empty in the $r \neq 0$ state, the cost $c_{node}$ of each node is added to the total cost $c$ of the current thread being traversed on line 25. Then, the algorithm continues to recursively call partition_dfs on each control successor of node on line 27.

When all branches (threads) under $r$ have been visited, the algorithm goes back to the $r = 0$ state on line 18 and cost is not calculated until the next top level fork node is reached. The reason cost is not calculated in the $r = 0$ state is due to the root thread being the only thread executing during this state. Then, the vector $C$, which represents the costs for each branch under $r$, is sorted in ascending order on line 19 before passing to the load_balance procedure. The load_balance procedure sorts the partitions based on their current associated costs. It then distributes each branch in $C$ to the partition with the least cost in $P$. This is repeated until all branches in $C$ have been distributed. Once load_balance returns to partition_dfs, $C$ is cleared and remains empty until a fork node is encountered.

### 6.3.2 Generating code from GRC

To execute the partitioned code in parallel, the code may be generated either as POSIX threads that require an OS to run, or as custom code for a dual-core Xilinx Microblaze platform that can run without an OS. The number of thread partitions that is generated will depend on the number of processor cores that the execution platform offers. These thread partitions are created only once throughout the lifetime of the program. Both the OS and non-OS implementations are based on the same principle that will be described next.

Generating code from GRC for execution in parallel is an incremental step from the implementation of the scheduling algorithm in described in Section 6.2.3. We will use
$r$ denotes the top level fork in the root thread

$c$ denotes the cost of the current branch being traversed

$c_n$ denotes the cost of the current node being visited

$C$ denotes a vector of costs associated with each branch under a fork

**Procedure partition_dfs(node)**

```
if node is visited then return

add node to the visited set

if node = the corresponding join of $r$ and $r \neq 0$ then
    return
else if node = fork and $r = 0$ then
    $r = node$
    foreach control successor $s$ of node do
        $c := 0$
        add $c$ into $C$
        partition_dfs(s)
    end
    $r := 0$
    sort $C$ in ascending order
    load_balance($C$)
    empty $C$
    return
end
if $C \neq \emptyset$ then
    $c := c + c_{node}$
end
foreach control successor $s$ of node do
    partition_dfs(s)
end
end
```

$p$ denotes the total cost of the partition

$P$ denotes a vector of total costs associated with each partition

**Procedure load_balance(C)**

```
foreach branch cost $c \in C$ do
    sort $P$ in ascending order
    assign the branch corresponding to $c$ to the first partition $p \in P$
    $p := p + c$
end
```

Figure 6.14: The distribution algorithm
the same sketch\(^1\) example presented in Fig. 6.10 to highlight the incremental changes. In Fig. 6.15, the sketch example has been divided into two partitions to execute on two processors. The shaded nodes execute on CPU2 and the non-shaded nodes execute on CPU1. The generated code corresponding to this partitioned GRC is presented in Fig. 6.16. Compared to the sequentially executed version in Fig. 6.12, there are a few additions in Fig. 6.16:

- Coordination and synchronization between processors have been added.

- There is more than one cyclic executive such that each partition executes within its own cyclic executive.

Initially, only the first processor executes the root thread. The first processor therefore, is appointed as the master processor. The master processor is responsible for coordinating the slave processors by instructing them to start executing when the root thread forks. To allow multiple processors executing from the same binary, the reactive function of an Esterel program is translated into a reentrant function. The reactive function is then called from each processor. Subsequently, processors could share the same address space and communication between processors can be implemented using shared memory.

The reactive function, esterel\_module, requires only one argument to indicate whether it is executing on a master processor or one of the slave processors. The slave processors are immediately blocked on line 4 until the master processor instructs them to start.

\(^1\)Note that we present a more abstracted example than the example in Fig. 6.13 so as to be able to present the generated code (see Fig. 6.16) within a one page limit.
int esterel_module(int isSlaveCPU) {
    if (isSlaveCPU) {
        WAIT_FORK:
        wait_fork(pc); // Blocking
        goto *pc;
    }
    thread_1.pc = 0;
    thread_2.pc = 0;
    lock_S = 3; // Three potential emitters
    // ...
    fork_cpu1(&&CPU1);
    goto CPU0;
}
CPU1:
    term_p1 = 0;
    do {
        term_1 = thread_1();
        term_2 = thread_2();
        term_p1 = term_1 | term_2;
    } while (term_p1 == INFINITY);
    join_cpu1();
    goto CPU0;
CPU0:
    term_p0 = 0;
    do {
        term_3 = thread_3(); // Definitions of thread_3() and thread_4() |
        term_4 = thread_4(); // are omitted to conserve space
        term_p0 = term_3 | term_4;
    } while (term_p0 == INFINITY);
    join_all(); // Blocking
    term_0 = term_p0 | term_p1;
    // ...
}
int thread_1() {
    if (thread_1.pc)
        goto *thread_1.pc;
    // ...
    RESUME:
    if (lock_S) {
        thread_1.pc = &&RESUME;
        return INFINITY;
    } else {
        // Inside the guarded body
    }
    // ...
}
int thread_2() {
    if (thread_2.pc)
        goto *thread_2.pc;
    // ...
    lock_S = 0;
    // ...
    END:
    thread_1.pc = &&END;
}

Figure 6.16: The generated code of the partitioned sketch example
When the master processor reaches the fork on line 11, it instructs the slave processors to begin execution by sending the starting addresses of the partitions allocated to the slave processors. For the master processor, it continues at the label CPU0, while the slave processor begins from the label CPU1. Both processors initialize their corresponding termination codes before the loop for the cyclic executive is entered.

The cyclic executive of each partition runs independently of each other at their own pace. Like the sequential version in Fig. 6.12, the termination code of each thread have to be combined when threads join. However, combining is a two step process in Fig. 6.16: once at the end of each partition on lines 18 and 27, and a second time after processors join on line 30. To synchronously join the two processors, a rendezvous is set up on line 29. This line blocks the master processor until all slave processors are ready to join. In this case, the single slave processor calls `join_cpu1` on line 20 to indicate that it is ready to join.

To illustrate how the semantics of instantaneous broadcast is preserved when threads execute in parallel (on different cores), the consumer thread of the signal $S$ has been deliberately scheduled before the potential emitters of $S$. More importantly, the example in Fig. 6.16 also demonstrates the elegance of run-time signal resolution working across processors. When the master processor sends the starting address to the slave processor on line 11, the slave processor has a head-start of a few instructions while the master processors is being held back by the overhead of coordinating the slave processor. The master processor starts executing the third thread shortly after the slave processor started executing the first thread. At this point, lock$_S$ is locked when the first thread attempts to read $S$ on line 38 and the first thread returns with a termination code of infinity. Assuming $S$ is not emitted by either the third or the fourth thread, lock$_S$ gets decremented twice. Let us also assume the second thread does not emit $S$ when the slave processor executes it and lock$_S$ becomes zero. The cyclic executive on line 19 detects that the first thread has been locked and reschedules it. Since lock$_S$ is now released, the first thread is able to successfully terminate and join the other threads.

We summarize the static load distribution approach by highlighting the following salient features:

- Unresolved signals are protected from premature access by global signal locks in the generated code.

- Scheduling of threads within partitions is managed by multiple cyclic executives executing on each processor.

- Forking and joining of threads that are allocated to different processors is coordinated by the master processor. The slave processors wait until the master processor instructs them to start.
• All processors execute from the same compiled binary and have different starting addresses within the binary.

While the static load distribution algorithm attempts to produce balanced load using the heuristic in Fig. 6.14, the estimated load can still vary significantly at run-time due to parts of the program react to inputs from the environment. As environment cannot typically be pre-determined at compile time, the load at run-time cannot in general be precisely calculated statically. In order to tackle this problem, the dynamic load distribution approach is proposed and discussed next.

6.4 Dynamic load distribution

Traditionally, Esterel compilers sequentialize threads using a topological sort [10]. This approach provides little room for executing threads in parallel on a multicore architecture and parallelizing threads in a balanced manner is difficult. This is due to the challenge of accurately estimating the load on each core, and the difficulty of modeling environment non-determinism that makes static load distribution an approximation at best.

The dynamic load distribution approach refines the cyclic executive in Fig. 6.1 of Section 6.1 by converting the reactive function into a distributed reentrant function. The reentrancy nature of the function allows it to be safely called from multiple cores simultaneously.

The most salient feature of the dynamic load distribution approach is the ability to freely move a thread from one core to another when a thread is rescheduled. The two distribution rules of the static approach described in Section 6.3 do not apply to the dynamic approach. The key to the difference is that the cyclic executive is globally shared by all cores using the dynamic approach in contrast to the distributed cyclic executives used by the static approach. Using the static approach, distributing threads without keeping the hierarchical structure of the threads intact would require performance costly synchronization between cores for starting a fork in addition to joining threads. In contrast, due to the global vector $T$ (line 1 in Fig. 6.17) being collectively managed by all cores, distribution of threads requires little overhead and is highly dynamic. This means threads can be distributed in any arbitrary way, as long as a parent thread is suspended when it is forked and one of its child threads resumes it when the child threads join. See Section 6.1 for details of how the hierarchical thread structure is preserved.

The algorithm presented in Fig. 6.17 takes advantage of the flexibility of dynamic load distribution by distributing a thread to a core whenever a core becomes idle. In Fig. 6.17, the distributed version of the reactive function requires an argument to identify whether the function is called from the master processor. The master processor is only responsible for starting each tick at the start of the root thread and return from the root thread at
$T$ denotes a vector of active threads as candidates to be scheduled

```plaintext
function reactive_function(isMaster)
if isMaster = true then
    foreach signal $s$ shared by threads in $T$ do
        $n_s :=$ the number of potential emitters of $s$
    end
    start executing the root thread
else
    while $T = \emptyset$ do
        do nothing
    end
end
forever
while $T \neq \emptyset$ do
    select and remove the first thread $t \in T$
    $\text{term\_code} := t()$
    if $\text{term\_code} = \infty$ then
        add $t$ to the back of $T$
    end
end
if isMaster = true then
    continue to execute the root thread
else
    while $T = \emptyset$ do
        do nothing
    end
end
end
```

Figure 6.17: The reentrant version of the high level cyclic executive with run-time signal resolution
the end of each tick. While the master processor starts executing the root thread on line 7, the slave processors remain idle on lines 9∼11 busy-waiting for the thread vector $T$ (global to all cores) to become populated. As soon as the root thread forks, all processors enter the cyclic executive on line 14. As the processors execute each thread in $T$, they interact with the cyclic executive in the following ways:

**Forks into more threads** — saves the executing thread in a buffer $F$, then add all child threads of the fork to the back of $T$. To ensure all child threads terminate synchronously, each child thread has to check the number of remaining child threads that are alive before terminating. The last child thread to terminate is responsible to resume and reschedule its parent thread by removing the parent from the buffer and adding it to $T$.

**Attempts to read a signal** — returns $\infty$ if the attempt to read failed due to unresolved signal when $n_s \neq 0$. Otherwise, the thread proceeds to read the signal and continue execution normally.

**Emits a signal** — emits the signal $s$ and reset the number of potential emitters by letting `$n_s := 0$'.

**Rules out emission of a signal** — removes the thread as a potential emitter of $s$ by decrementing $n_s := n_s - 1$.

As long as $T$ is not empty, the processors will continue to execute within the cyclic executive and threads are able to execute on any processor and in any order. This highly dynamic execution scheme highlights the elegance and flexibility of run-time signal resolution and the new load distribution approach. Due to the dynamicity, which processor will be the last processor to exit the cyclic executive cannot be pre-determined. To ensure that the root thread is always executed on the master processor, line 21 is required to look for the master processor. The slaves processors will enter idle state waiting for the next fork to occur in the root thread while the master processor executes the root thread.

The next subsection will use a small example to visualize how the distributed reactive functions execute in parallel.

### 6.4.1 High level representation of dynamic thread distribution

The implementation of the distributed version of the reactive function consists of two aspects: (1) thread distribution and (2) run-time signal resolution. We will first describe the implementation of thread distribution using the sketch example and describe the implementation of run-time signal resolution in the next subsection. The GRC representation of the example is exactly the same as Fig. 6.11 and it is reproduced in Fig. 6.18.
with colours to aid the discussion of thread distribution. The thread vector $T$ in Fig. 6.17 is implemented as a thread queue using a ring-buffer. The ring-buffer is initially empty and only the first processor is executing the root thread. The second processor remains idle awaiting threads to be inserted into the ring-buffer.

Let us assume the sketch example is executing on a dual-core processor. The initial state is depicted in Fig. 6.19(a). The white segmented ring in Fig. 6.19(a) represents the ring-buffer while the two blocks depicted at the center of the ring-buffer represent the statuses of the processor cores. The gray CPU2 label indicates that the second processor is currently idle. When the root thread reaches the fork node in Fig. 6.18, it spawns four threads by adding those threads to the ring-buffer and saves the address of the root thread in a separate buffer. The buffer containing the root thread is depicted as the square labeled $T_0$ in Fig. 6.19(b). The populated ring-buffer consists of $T_1$ (green), $T_2$ (yellow), $T_3$ (cyan) and $T_4$ (gray). The colour of each thread correspond to the nodes in Fig. 6.18 shaded with the same colour.

While the first processor completes the fork process, the second processor fetches $T_1$ from the ring-buffer and starts executing. Shortly after the second processor started executing, the first processor fetches $T_2$ from the ring-buffer and starts executing in parallel. As the two processors execute, two threads remain in the ring-buffer, as illustrated by Fig. 6.19(c). Assuming $S$ is not emitted by $T_2$, $S$ remains locked and $T_1$ is unable to proceed as it has been blocked from reading $S$. The cyclic executive on the second processor detects the blocked thread and adds $T_1$ to the back of the ring-buffer.

The state of the ring-buffer is now represented by Fig. 6.19(d). The ring-buffer consists of $T_3$, $T_4$ and $T_1$ and the second processor is about to fetch another thread from the ring-
Figure 6.19: The time-line of the state of the thread queue and processors
buffer. Then, the second processor fetches \( T_3 \) followed by the first processor fetching \( T_4 \) from the ring-buffer. The ring-buffer now has only \( T_1 \) as the two processors begin to execute \( T_3 \) and \( T_4 \), as illustrated in Fig. 6.19(e). Assuming \( S \) is not emitted in either \( T_3 \) or \( T_4 \), \( T_4 \) terminates normally followed by \( T_3 \) shortly after. Since the second processor completed \( T_4 \) before the first processor, it fetches \( T_1 \) from the ring-buffer and starts to execute. As the ring-buffer is now empty, the first processor has nothing to fetch and becomes idle. In Fig. 6.19(f), the second processor is the only active processor executing \( T_1 \). With all the potential emitters completed without emitting \( S \), \( T_1 \) is now unblocked and finishes normally.

When the second processor executes the join, it recovers the address of the root thread from the buffer saved during the fork and sends the address to the first processor. Then, the first processor is waken up by the second processor and resumes the root thread. Once the first processor takes over the control, the second processor becomes idle again until the next fork. Finally, the ring-buffer and the processors enter the same state depicted by Fig. 6.19(a).

The next subsection will describe the implementation of dynamic thread distribution and run-time signal resolution.

### 6.4.2 Implementation of dynamic thread distribution

Implementing an efficient thread queue is essential in order to not impede any performance gain. Typical operations on the queue involve appending to the tail of the queue and fetching from the head on a regular basis. Thus, implementing the queue as a ring-buffer would be more efficient than a linear buffer. Appending to the queue requires the following micro-steps:

- check if the queue is full
- place a thread after the item currently pointed by the tail pointer
- update the tail pointer

And popping a thread from the queue requires the following micro-steps:

- check if the queue is empty
- retrieve the thread pointed by the head pointer
- update the head pointer

Each micro-step would take a few instructions to implement in software. As a typical Esterel program frequently creates and destroys threads, accessing the thread queue on a regular basis would generate considerable overhead.
To minimize the overhead of managing the thread queue, the queue has been implemented in hardware. The hardware allows efficient mutual exclusive reading and writing to the queue. Simultaneous access to the queue will result in some cores being briefly blocked up to $n$ clock cycles, where $n$ equals to the number of cores accessing the queue simultaneously. The counting semaphore used for implementing run-time signal resolution is also implemented in hardware. The hardware based counting semaphore also works as a hardware mutex for atomic access to shared variables. Both hardware units are accessed through the memory bus. The interaction with the hardware is best illustrated using the example in Fig. 6.20 generated from the sketch example in Fig. 6.18.

In the generated code, `FIFO_BASE`, `MUTEX_SETKEY`, `MUTEX_LOCK` and `MUTEX_UNLOCK` are memory mapped registers. These registers provide access to the thread queue and the mutex in the hardware. A thread can be added to the thread queue by writing the starting address of the thread to `FIFO_BASE`. A thread can be retrieved from the thread queue by reading `FIFO_BASE`. Reading this register effectively removes a thread from the queue. Similarly, the mutex hardware can be accessed by reading or writing to its memory mapped registers. The mutex hardware is implemented as a map that stores key-value pairs representing pairs of lock and its semaphore count. The mutex has three registers for setting the protected memory address, setting the semaphore count, decrementing and retrieving the semaphore count. The first register is `MUTEX_SETKEY`. It is a write-only register for setting the memory address to be protected. The second register is `MUTEX_LOCK`. It is also a write-only register. It initializes the counting semaphore to the value written to the register. The third register is `MUTEX_UNLOCK`. It is a read-write register that decrements the semaphore count when a memory address is written to it. The current value of the counting semaphore can be obtained by reading the `MUTEX_UNLOCK` register.

For example, the content of the hardware thread queue and the mutex are presented in Fig. 6.21. When the master processor reaches line 13 in Fig. 6.20, the thread queue contains four threads and the hardware mutex holds a single lock. The address of `lock_S` stored in the mutex maps to a value of three. This value is either decremented by the potential emitters (line 35 in Fig. 6.20) or reset to zero when the signal is emitted. Resetting a semaphore count works the same way as initializing it with a value, such as line 6 in Fig. 6.20. The size of the hardware queue and the mutex is parameterizable and are limited by the hardware resources available.

Let us assume the sketch example is executing on a dual-core processor and follow the same execution trace as Fig. 6.19. Both cores start executing by calling the `reactive function – esterel_module`. Each core is distinguished by its unique ID, which is passed as an argument. The first core has the ID zero and acts as the master processor while the second core becomes the slave. While the first core initializes the counting semaphore of $S$, the second core is immediately blocked upon reading the `FIFO_BASE` register on line 3
int esterel_module(int coreID) {
    if (coreID)
        goto *FIFO_BASE;
    else {
        MUTEX_SETKEY = &lock_S;
        MUTEX_LOCK = 3; // Three potential emitters
    }
    _thread_0_count = 4;
    FIFO_BASE = &THREAD_1;
    FIFO_BASE = &THREAD_2;
    FIFO_BASE = &THREAD_3;
    FIFO_BASE = &THREAD_4;
    goto *FIFO_BASE;
    THREAD_1:
    // ...
    RESUME:
    MUTEX_SETKEY = &lock_S;
    if (MUTEX_UNLOCK) {
        FIFO_BASE = &RESUME;
        goto *FIFO_BASE;
    } else {
        // Inside guarded body
    }
    // ...
    MUTEX_SETKEY = &_thread_0_count;
    MUTEX_LOCK = 1;
    _thread_0_count --;
    MUTEX_UNLOCK = &_thread_0_count;
    if (_thread_0_count)
        goto *FIFO_BASE;
    else
        goto THREAD_0;
    THREAD_2:
    // ...
    MUTEX_UNLOCK = &lock_S;
    // ...
    MUTEX_SETKEY = &_thread_0_count;
    MUTEX_LOCK = 1;
    _thread_0_count --;
    MUTEX_UNLOCK = &_thread_0_count;
    if (_thread_0_count)
        goto *FIFO_BASE;
    else
        goto THREAD_0;
    THREAD_3:
    // Omitted to conserve space
    THREAD_4:
    // Omitted to conserve space
    THREAD_0:
    // ...
    if (coreID == 0)
        // ...
    else {
        FIFO_BASE = &THREAD_0;
        goto *FIFO_BASE;
    }
}

Figure 6.20: The sketch example illustrating thread queue access in the generated code
due to an empty thread queue. The first core then starts executing the root thread from line 9. This line, which corresponds to the fork node in Fig. 6.18, begins by initializing the thread count and adding the starting address of each thread to the thread queue. The implementation of the fork process differs slightly to the high level representation of thread distribution in Fig. 6.19. Instead of saving the address of the root thread into a separate buffer, a thread count is initialized with the number of child threads being spawned. Since the continuation address when the root thread resumes is known at compile-time, the address is simply referenced by the label on line 49. However, the thread count is required to track the number of active threads spawned by the fork.

As soon as a thread is added to the queue, the second core becomes unblocked and attempts to fetch a thread from the queue on line 3. As the second core is accessing the queue in parallel to the first core, one of them will be blocked by the other for a clock cycle. Assuming the second core gets the access to the queue, it successfully fetches $T_1$ and starts executing from line 34. Meanwhile, the first core completes the fork process and fetches $T_2$ from the queue. Assuming $S$ is not emitted by $T_2$ executing on the first core, the semaphore count of $S$ is decremented by $T_2$ on line 35 and two potential emitters remain. On line 18, the second core attempts to read $S$ but fails. It then adds $T_1$ back to the queue on line 18. At this point, the current state of the queue corresponds to Fig. 6.19(d). The second core then fetches $T_3$ and starts executing from line 46. Shortly after the second core started executing $T_3$, $T_2$ executing the first core terminates normally and decrements the thread count on line 39. As three more threads are still active, the first core fetches $T_4$ from the queue and starts executing from line 48. The current status of the queue is now represented by Fig. 6.19(e).

Assuming neither $T_3$ nor $T_4$ emit $S$, no more potential emitters remain and $S$ is unlocked. The second core completes $T_4$ normally and fetches $T_1$ from the queue and starts executing from line 17. In the meantime, the first core also finishes executing $T_3$ and it sees one thread is still active indicated by $\_\_thread\_0\_count$. The first core attempts to fetch another thread but gets blocked by the empty queue. The states of the cores and the queue now correspond to Fig. 6.19(f).
As $S$ is now unlocked, $T_1$ successfully completes and decrements the thread count. Since no more thread is active, the second core jumps from line 32 to line 49 and resumes the root thread. However, the root thread has to execute on the first core, the second core passes the control to the first core by adding the resumption address to the queue on line 54. Finally, the first core successfully takes over control and continues from line 52.

We summarize the elegance and the flexibility of the dynamic thread distribution approach by highlighting the following salient features:

- Unresolved signals are protected from premature access by signal locks implemented as hardware counting semaphores.

- Scheduling of threads are collectively managed by a single cyclic executive shared by all processors. There is little overhead from moving a thread from one processor to another as other processors are able to fetch threads from the globally shared hardware queue directly.

- Forking threads can be performed by any processor whenever the processor’s assigned thread forks. The newly spawned child threads are dynamically added to the global thread queue and the parent thread is suspended by leaving it out from the thread queue. Joining threads is done by whichever processor that happens to execute the last terminating child thread. The resume address of the parent thread is known at compile-time and the parent thread is woken up when scheduled by adding the parent thread to the queue.

- All processors execute from the same compiled binary and have different starting addresses within the binary.

### 6.5 Correctness of run-time signal resolution

The run-time signal resolution approach is correct as long as the Esterel program is known a priori to be causal [8]. Causality issues stem from instantaneous read-write cycles between signal readers and writers. such as the program `present A else emit A end`. The program reads, if $A$ is absent, immediately emit $A$. The `present` statement contradicts with the `emit` statement, as a signal cannot change its status within a `tick`. The run-time signal resolution approach assumes and relies on the program being causal. Thus, the formalization of the correctness of our scheduling approach in Theorem 1 assumes the program to be causal.

To aid readability, Fig. 6.1 has been reproduced as Fig. 6.22. Based on the algorithm in Fig. 6.22, the following axioms hold:
1 $T$ denotes a vector of active threads as candidates to be scheduled
2 $t$ denotes a sequential composition of Esterel statements represented as a function
3 $S$ denotes a set of unresolved signals shared by threads in $T$
4 function reactive_function
5 Populate $S$ with signals shared by $t \in T$
6 add the root thread to $T$
7 while $T \neq \emptyset$ do // start of the cyclic executive
8 select and remove the first thread $t \in T$
9 term_code := $t()$
10 if term_code = $\infty$ then
11 add $t$ to the back of $T$
12 end
13 end
14 return term_code
15 end

Figure 6.22: A cyclic executive for run-time signal resolution

**Axiom 1.** For any causal Esterel program, the cyclic executive in Fig. 6.22 schedules every active thread at least once in an iteration.

**Axiom 2.** All signals in $S$ cannot be tested until their statuses have been resolved and removed from $S$. Premature test of a signal is not possible due to $S$ being reinitialized by the scheduling algorithm in Fig. 6.22 (line 5) at the start of each tick.

**Axiom 3.** Each scheduled thread completes by returning a termination code. Threads that are blocked due to an unresolved signal will return a termination code of $\infty$, or a finite positive integer otherwise. Threads that return a termination code of $\infty$ will be rescheduled in the next iteration of the cyclic executive.

**Axiom 4.** Execution of a thread during an iteration results in one of the following outcomes:

1. One or more shared signals are resolved to be present due to emissions, or absent when all of their potential emitters collectively agree that those signals can no longer be emitted in that tick. In either case, the number of unresolved signals, $|S|$, will be decremented by the number of resolved signals.

2. The thread gets blocked while trying to read a signal that is unresolved. That thread will be rescheduled in the next iteration of the cyclic executive. $|S|$ remains unchanged in this case.

3. The thread gets removed from $T$ because it neither emits nor tests a signal in $S$. Consequently, the thread successfully completes a tick and $|S|$ remains unchanged.
Based on the axioms above, we will now prove that the number of unresolved signals decrease monotonically as described in Theorem 1.

**Theorem 1.** Let $C$ be a causal Esterel program. Further, let $T$ be the set of parallel threads in $C$, and $S$ be the set of signals shared (emitted and tested) between two or more threads in $T$. Then, for any execution order of all threads $t_i \in T$ ($i \in \mathbb{N}$) in each tick, the number of signals in $S$ that are unresolved will decrease monotonically, and eventually converge to zero after at most $|S|$ iterations of the cyclic executive shown in Fig. 6.1.

**Proof.** The proof for Theorem 1 starts by showing that the number of unresolved signals $|S|$ will decrement by at least once during each iteration using induction.

### 6.5.1 Base case

Consider the case where $S = \{s_1\}$ is shared among a set of threads, $T = \{t_0, t_1, \ldots, t_M\}$. There are two possible scenarios:

- If $t_i$ ($i \in \{0, 1, \ldots, M\}$) only tests but does not emit $s_1$, $|S|$ remains the same (Outcome 2 under Axiom 4).
- If $t_i$ is a potential emitter of $s_1$, then by Axiom 2, there are two possibilities:
  1. Signal $s_1$ is emitted.
  2. Thread $t_i$ rules itself out as a potential emitter of $s_1$.

Once the cyclic executive schedules all threads in $T$, either $s_1$ will be emitted (resolved as present) or the number of potential emitters of $s_1$ will be zero and $s_1$ will not be emitted (resolved as absent). Therefore, the number of unresolved signals will reduce to zero after at most one iteration of the cyclic executive.

### 6.5.2 Hypothesis

Assume that Theorem 1 holds for the case where $S = \{s_1, s_2, \ldots, s_j\}$.

### 6.5.3 Inductive step

Now, consider the case where $S = \{s_1, s_2, \ldots, s_j, s_{j+1}\}$ is shared among a set of threads, $T$. After $j$ iterations of the cyclic executive, at least $j$ signals would have been resolved, leaving at most one unresolved signal, say, $s_u$. Then, in the $(j+1)$th iteration of the cyclic executive, the execution of thread $t_i$ will again result in one of two scenarios as before:

- Thread $t_i$ gets blocked while attempting to read $s_u$ (Outcome 2).
• Thread $t_i$, by Axiom 2, either:
  
  1. emits $s_u$; or
  2. rules itself out as a potential emitter of $s_u$.

If $s_u$ gets emitted, all signals are resolved and $|S|$ becomes zero. Otherwise, threads in $T$ will rule themselves out as potential emitters.

At the end of the $(j+1)$th iteration, $s_u$ will either be emitted, or have all its potential emitters ruled out. It cannot happen that $s_u$ is still unresolved, since the statuses of all other signals in $S$ are already known. The first scenario can only perpetuate after the $(j+1)$th iteration if the test of $s_u$ itself determines its resolution (a contradiction of the causal assumption). Therefore, the cyclic executive will always resolve at least one signal in $S$ in each iteration. Since $|S|$ reduces by at least one over each iteration, the proof for Theorem 1 simply follows.

Due to Theorem 1 and Axiom 2, the semantics of instantaneous broadcast is preserved. The resulting behaviour of a program that resolves signals at run-time would exactly match a statically scheduled program.

### 6.6 Experimental results

To evaluate the proposed approaches, a set of mixed control and data dominated programs have been selected for benchmarking. These programs are presented in Table 6.1. The first column in Table 6.1 lists the names of the programs. The number of lines of Esterel code is shown in the second column and the third column shows the number of lines of C code. The C code implements host procedures that are called from Esterel. The fourth column shows the maximum number of threads that can potentially be executing in parallel.

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC (Esterel)</th>
<th>LOC (C)</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABIC</td>
<td>21</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MCA200</td>
<td>4956</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>WW (Estbench)</td>
<td>1087</td>
<td>676</td>
<td>22</td>
</tr>
<tr>
<td>WW09</td>
<td>317</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>LiftController</td>
<td>272</td>
<td>230</td>
<td>12</td>
</tr>
<tr>
<td>LZSS</td>
<td>42</td>
<td>462</td>
<td>5</td>
</tr>
<tr>
<td>Life</td>
<td>74</td>
<td>105</td>
<td>4</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>168</td>
<td>160</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.1: A list of benchmarking programs
The first program ABIC is the smallest example with only 21 lines of code. This program was created to test how well our approach works with small Esterel programs that contain no data computation. The second program, MCA200, is taken from the Estbench benchmark suit [45]. It is a program that implements a shock absorber controller. It is also the largest control dominated Esterel example listed here, with no data computation. WW (Estbench) is a wristwatch example also taken from [45]. It is the largest program of all with considerable amount of data computation through C function calls. WW09 is an alternative implementation of wristwatch developed from scratch using purely control statements. This example does not contain any data computation. LiftController is a design based on Vahid and Givargis’s example [50]. The design has both control and data parts. LZSS is a text archiver using the Lempel-Ziv Storer Szymanski (LZSS) algorithm. This program contains the largest amount of data computation in C code of all the examples shown here. The Life example is a program that simulates the evolution of the program determined by its initial state. It is adapted from the MPI version [51] into Esterel. This example performs large amounts of arithmetic computation over a two-dimensional array. Mandelbrot, adapted from [52], is a program that computes a mandelbrot set. The data computation involves floating point calculation and this type of computation is used to simulate a typical DSP application.

In the following subsections, two types of implementation will be benchmarked. We will first present the evaluation of the OS-based implementation using POSIX threads in Section 6.6.1, followed by an implementation using embedded multicore without an OS in Section 6.6.2.

6.6.1 OS-based implementation using POSIX threads

To test the POSIX thread based implementation running on a commodity multicore, a desktop PC powered by a 2.4GHz Intel Core 2 Quad processor was used. The programs were compiled with GCC 4.5.0 and executed in Linux. The performance of the programs running on the desktop were measured using a pair of `clock_gettime()` C library calls. The time measured corresponds to how long each program took to complete the given input trace.

The results for static load distribution reveal that control dominated programs are not amenable to speedup on multicore platforms. The execution time of these programs are shown in Table 6.2. From the table, one can notice that control-dominated programs, do not benefit from the POSIX thread implementation running with an OS. The lack of performance gain can be attributed to the following:

- The Esterel scheduling overhead outweighs the actual computation performed within each Esterel thread. This happens when the Esterel threads found in a program are
Table 6.2: Performance comparison on PC

<table>
<thead>
<tr>
<th>Examples</th>
<th>1 core</th>
<th>2 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABIC</td>
<td>3.5ms</td>
<td>4ms</td>
</tr>
<tr>
<td>WW (Estbench)</td>
<td>357ms</td>
<td>410ms</td>
</tr>
<tr>
<td>WW09</td>
<td>349us</td>
<td>347us</td>
</tr>
<tr>
<td>LiftController</td>
<td>2.57ms</td>
<td>3.06ms</td>
</tr>
</tbody>
</table>

Figure 6.23: Performance comparison on PC (the lower the better)

- Load distribution is highly uneven.

The LZSS example, a heavily data dominated application has been selected as the main benchmark to run on the PC. LZSS has a parameter that sets the dictionary size of a text archiver, which uses the Lempel-Ziv Storer Szymanski algorithm [53]. The size of the dictionary affects the length of the data computation within a tick. The dictionary size can be practically set between 64 bytes to 8192 bytes. Beyond 8 kilobytes, the program would take too long to complete within a reasonable time-frame. The number of cores LZSS uses can be controlled by setting the number of partitions used in the compiler. Using the quad-core system described in Section 6.6.2, LZSS can be distributed up to a maximum of four partitions. The plot in Fig. 6.23 shows that LZSS will benefit from multicore execution as the dictionary size increases. When running with all four cores, the maximum speedup achieved in the experiment is 3.7 times compared to a single core.

6.6.2 Using an embedded multicore architecture

To evaluate the performance of executing Esterel on a multicore without an OS, an embedded multicore architecture has been developed for benchmarking. The architecture
consists of four Xilinx Microblaze processors. Each core has its own local on-chip memory for fetching instructions and storing local data. Typically, the instruction memory port and the data memory port on a Microblaze processor are both connected to the same dual-port, on-chip memory. For a dual Microblaze system, the dual-port memory can be used as a shared memory between the two cores for communication purposes. When the on-chip memory is connected to the Local Memory Bus (LMB), accessing the memory takes only a single clock cycle. This still holds when both of the memory ports are being accessed at the same time. The fast memory access is essential for a multicore architecture. Both cores can access the local memories and the shared memory within one processor clock cycle. For more than two cores, the dual-port memory needs to be multiplexed. As an example, a quad-core system using the multiplexed design is illustrated in Fig. 6.24.

The quad-core system consists of separate Local Memory Buses for each core. The local memory and the shared memory are separated in the address space as illustrated by Fig. 6.24(b). On processor start up, both the local memory and the shared memory in the system are programmed with the same program executable. Each core fetches instructions and can accesses local data from its own local memory, depicted as IM/DM in Fig. 6.24(a). The cores are multiplexed in round robin fashion when simultaneous access to the shared memory is requested. The maximum number of clock cycles to access the shared memory directly corresponds to the number of cores requesting for access.

In order to benchmark the scalability of the architecture, a Microblaze multicore simulator was designed by adapting [54]. The simulator is able to simulate an arbitrary number of cores. The simulator is used to benchmark a hypothetical quad Microblaze
system, where synthesizing this architecture would put too much constraint on the amount of on-chip memory available for each core.

### 6.6.3 Performance of static load distribution

To find out how well the static load distribution works for an embedded system without an OS, the same examples were executed on a Xilinx dual-core MB system without any OS. The performance was measured using hardware clock counters. The counters are controllable from their associated processors. Both counters are synchronized with the processor clock. Performance measurement requires the use of only the counter on the master processor. The counter on the slave processor is only used when measuring the load distribution.

The performance criteria is the reaction time of the tick length. With the precision offered by the counters, the reaction time of each program can be measured with an accuracy of the exact number of processor clock cycles it took to execute. The counter on the master processor is started just before it enters the reactive function, and stopped right after the reactive function returns. The difference between the two clock counts gives the reaction time of the current tick. The reaction times over the number of ticks taken to complete the input trace is then summed up. The average case reaction time (ACRT) can then be obtained by dividing the sum by the number of ticks.

Fig. 6.25 compares the ACRT of the programs when running on a single core with that on a dual-core (labeled with DMB). Having seen the results of the POSIX thread based implementation in Table 6.2, it is not surprising to see ABIC and Wristwatch (Estbench) performing worse. With bigger threads, examples such as Wristwatch09 and LiftController were able to perform slightly better with two cores.

Again, the LZSS example is benchmarked with a range of dictionary sizes. The trend of the ACRT in Fig. 6.26 closely resembles the plot in Fig. 6.23. Overall, the speedup obtained from multicore execution without an OS is slightly better than execution with an OS. The LiftController example showed speedup in Fig. 6.25, whereas the same example ran slower in Table. 6.2. The overhead from the threading library and the OS has caused the LiftController example to run slower with an OS.

### 6.6.4 The load balance of static load distribution

To evaluate the load balance, the maximum, minimum, and average processor idle time between each fork-join pair on the dual-core Microblaze were measured. The percentage shows the relative idle time of a core when a fork occurs. The results are presented in Table 6.3. The most balanced program is LZSS due to its computation-intensive nature. The data can be segmented perfectly for balanced execution in parallel. The least balanced
Figure 6.25: Performance comparison on dual-core Microblaze (lower the better)

Figure 6.26: ACRT comparison of LZSS only on dual-core Microblaze (lower better)
Table 6.3: Comparison of processor idle time in %

<table>
<thead>
<tr>
<th>Examples</th>
<th>Max (%)</th>
<th>Min (%)</th>
<th>Average (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABIC</td>
<td>52</td>
<td>48</td>
<td>50</td>
</tr>
<tr>
<td>WW (Estbench)</td>
<td>72</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>WW09</td>
<td>49</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>LiftController</td>
<td>96</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>LZSS</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

program is LiftController. The data computation in this example is concentrated in one thread, while the remaining threads only perform control. WW (Estbench) and WW09, having more threads, were able to achieve a single digit percentage in processor idle time in the best case, but achieved 72% and 49% in the worst case respectively. On average, WW (Estbench) has a processor idle time of 40%, while WW09 is slightly better with 34%. ABIC, the smallest example, has about 50% in processor idle time in all cases.

These results illustrate that achieving speedup with multicore execution of Esterel is dependent on several factors. One of the key factors is the computation to communication ratio which is determined by the amount of dependency between threads on the different cores. The heuristic guided distribution tries to minimize this signal dependency. However, another important factor is the effect of such distribution on the processor idle time. As can be observed, the distribution is not designed to minimize the amount of time a processor remains idle since it is heuristically guided. Improving this will be a key to achieving more consistent and better speedup of the distribution algorithm. By intuition, if a core becomes idle, it should ideally steal more work to do. This would require load distribution to be done at run-time. Hence, the dynamic load distribution approach has been developed. The evaluation of this approach is described next.

6.6.5 Performance benchmark of dynamic load distribution

The experimental results obtained from the dynamic load distribution approach is in big contrast to the static heuristic based load distribution. All benchmarked programs showed speedup across the board, as shown in Fig. 6.27.

The first example, Life, spawns only two threads throughout the program. Therefore, there are no results for tri-core and quad core experiments. It did, however, benefit from a dual-core with a performance gain of 99%. The LZSS and LiftController example showed interesting results for three cores compared to four cores. This ambiguity maybe explained as follows. There are four threads of identical load spawned by the program. When executed with three cores, three of these threads will be completed almost at the same time. The remaining thread will be picked up by whichever core completed its work first, leaving the other two cores idling. However, executing on four cores, all threads are evenly distributed and completed at the same time, resulting in minimal processor idle
time. All other examples show varying degrees of performance gains from two to three cores. On average, the ACRT performance increased by 57% with two cores, 82% with three cores, and 130% with four cores.

The worst case reaction time (WCRT) measured from the longest tick length during the execution of each program is shown in Fig. 6.28. The figure illustrates a similar trend compared to the ACRT. However, upon careful inspection of the results for the WCRT, one can still observe that the examples benefit more from four cores than three cores. The WCRT performance on average increased by 65% with two cores, 99% with three cores, and 163% with four cores.

Overall, all examples showed performance gains from multicore execution. Obviously, the data dominated examples benefited the most, while examples with less data computation did not gain as much. Interestingly, for a pure control program, if the program is large enough with many threads, the program may still have a good potential to perform better with a multicore processor. The MCA200 is a pure control program that nicely demonstrated significant benefits using the dynamic load distribution approach.

6.7 Discussion

A key to effective parallelization of Esterel programs is a scheme for resolving the status of signals at run-time. Another essential element is a method for load balancing that tries to minimize communication between the cores. This chapter introduced two approaches that
address both questions with a compiler that can either (1) statically distribute threads based on a greedy heuristic; or (2) dynamically distribute threads at run-time to reduce the processor idle times. These techniques highlight the novelty of the proposed approach.

In contrast to conventional scheduling techniques, our compiler introduces the concept of signal locks to resolve signals at run-time. A signal remains locked until it is resolved to be either present or absent in every tick. The locking mechanism introduced in this chapter can compute signal absence, and preserves reactivity and determinism of any causal Esterel program while doing so. The benchmarking results reveals that static load distribution is effective for parallelizing Esterel programs involving sufficient data computation.

While the static load distribution algorithm worked well for data dominated programs, its load distribution heuristics does not work so well for control dominated programs. The second approach presented in this chapter overcomes the problems by dynamically distributing the load at run-time. The experimental results have shown that the technique has improved the performance substantially. The benefits of the dynamic distribution are as follows:

1. The technique is capable of scaling across any number of processor cores, and is limited only by the hardware resources available.

2. The technique does not require complex timing analysis of the work load.

3. The processor idle time is less sensitive to environment non-determinism compared
to static distribution.

We have demonstrated the effectiveness of our techniques over a benchmark of control-dominated and data-dominated programs for parallel execution. The thread distribution techniques described in this chapter are able to distribute both the control code and data of Esterel. In contrast, the distribution technique introduced by Caspi et al. [25] replicates the control code across a network of processors, distributing only data.

Compared to [26], the effectiveness of their approach to parallelizing an Esterel program is unclear as parallelism is relatively coarse-grain compared to our approach. The distribution technique introduced in that work depends on the number of concurrent execution paths without data dependencies. As the threads within an Esterel program are tightly coupled, the effectiveness of their technique would be severely limited. Moreover, distribution in that work is achieved through OpenMP, which relies on an OS. Our approach will work with and without an OS.

An interesting work introduced by Ju et al. [28] described an approach to estimate worst-case reaction time (WCRT) of Esterel programs running on multiprocessors. They have also adopted the run-time signal resolution technique described in [29]. The contribution of that work is unclear as the estimated WCRT is only reported for multicore execution and did not compare to the single-core execution. In contrast, we have shown the effectiveness of our approach through a set of benchmarking programs executed on both single-core and multicore processors.

The run-time signal resolution technique described in this chapter is the key to effective distributed execution of Esterel. Despite the dynamic nature of the scheduling algorithm, Section 6.5 has proven that the approach is sound and correct. Without the flexibility of the scheduling algorithm, load distribution would be tightly coupled with the scheduling problem. Solving scheduling and load distribution at the same time would be far more complex, achieving speedups with parallel execution on multicore would be extremely difficult.
Conclusions and Future Work

Embedded systems are growing exponentially in complexity, with many embedded applications even relying on multi-core systems for achieving better power-performance trade-offs. For example, automotive engine control units are starting to use multi-core processors. In contrast to such developments, even today, many programmers are accustomed to writing embedded software in C, and manage concurrent tasks with an Operating System (OS). While C is a powerful general purpose language, it lacks concurrency. Relying on an OS to manage concurrent tasks not only increases the system execution overhead, but often produces inconsistent behaviours. The behaviour of the system is further complicated by the system’s interrupts, making verification a near impossible task.

The synchronous programming paradigm introduced an elegant way of writing concurrent, deterministic and reactive programs. Esterel, as a synchronous language, is no exception. The inherent concurrency of the language greatly enhances the intuitiveness of specifying concurrent tasks typically found in embedded systems. The formal semantics of Esterel has further enabled formal verification of the written program.

While Esterel has brought powerful features for specification of embedded systems, the language is notoriously difficult to compile. The complexity mostly revolves around the instantaneous synchronous broadcast of signals for communication between Esterel threads. To guarantee that any signal emitted within a tick is detected by its reader, the compiler must ensure signal writers are always executed before signal readers. Many compilation techniques have been explored in [55, 21, 10, 12]. Except [12], other com-
Conclusions and Future Work

Compilers implement Esterel using the pure software approach by compiling Esterel into C, sacrificing performance for flexibility. The intermediate approach first introduced in [11] maintains the flexibility of the pure software approach, and further improves the performance and the generated code size by designing a processor that accelerates the execution of Esterel. However, the processor has been designed to handle the majority of the Esterel constructs, severely limiting the maximum clock frequency the processor is able to operate at and the size of Esterel programs. Finally, none of these compilers are able to generate code for the ever increasing ubiquitous multi-core processors.

With the goal of achieving efficient execution of Esterel, this thesis has proposed solutions to compile Esterel for performance gains. This research has demonstrated the efficacy of architecture specific compilation for a processor tailored for efficient execution of Esterel, and has addressed the issue of compiling Esterel for multi-core processors.

In Chapter 4, the design of a simple, yet efficient processor has been presented. The processor, called STARPro, strikes a better balance between hardware complexity and performance scalability. STARPro accelerates the execution of Esterel with its thread management hardware unit and a deterministic preemption handling unit. Unlike KEP3a [12], STARPro does not maintain the Esterel semantics in hardware. Instead, the question of adhering to the semantics is cast as a compiler problem. This has resulted in much simpler hardware design, while achieving better performance and reasonable code size compared to [12]. STARPro has achieved on average 37% faster average case reaction time than KEP3a, and 38% faster worst case reaction time. However, the code size is on average 40% bigger than KEP3a, but still many times lower than coded generated from conventional Esterel compilers.

Chapter 5 addresses the questions surrounding compiling for STARPro. The compiler represents Esterel in an intermediate format called Unrolled Concurrent Control-Flow Graph with surface and depth (UCCFGsd). The UCCFGsd preserves the structure of the source completely, while at the same time, maintains a one-to-one mapping to the STARPro assembly instructions. This chapter has been dedicated to explain the translation process of each Esterel statement, and how threads are topologically sorted for scheduling at run-time in the thread management unit of STARPro.

The final question of compiling Esterel for multi-core has been addressed in Chapter 6. This chapter has first described a scheduling technique that allows threads to execute in any arbitrary order by resolving the signals at run-time. This dynamic scheduling approach is the first key to enable parallel execution of Esterel threads. The other key is to solve the question of thread distribution. The research was initially started by exploring the viability of executing Esterel on multi-core with a static distribution algorithm. The results of this first attempt have showed promise of speedups with large data dominated programs. The lack of performance gains for control dominated programs have been found
to be a load balancing problem. Subsequently, a dynamic load distribution approach has been presented that significantly improves the load balancing issue. By combining dynamic scheduling with a hardware thread queue, not only can threads now be executed in any order, but also be able to execute on any processor core and be distributed to any number of cores. This flexibility and scalability has resulted in speedups for both large control and data dominated programs. With two cores, the performance for average case reaction time has increased by 57% on average, 83% with three cores, and 130% with four cores.

This research has so far achieved efficient execution of Esterel for different performance requirements. For extremely compact code size and fast reaction time in a small embedded system, the STARPro suits this need very well. For programs with heavy computations and large amount of parallel threads, this thesis has also provided an approach to execute such programs efficiently on a multi-core processor. While the solutions proposed in this thesis worked well in the current form, the next section will describe potential future directions this research can work towards.

7.1 Future work

This thesis has thus far provided solutions to achieve more efficient execution of Esterel, several areas of this research still have room for improvement. This section will elaborate on each area that could be worked on in the future.

7.1.1 Handling data computation

The STARPro compiler discussed, in Chapter 5, is currently unable to handle host procedure calls written in C. It is up to the programmer to manually translate from C to STARPro assembly instructions. This is the consequence of compiling Esterel straight to assembly instructions. This gap can be filled by developing a C compiler backend for STARPro. LLVM [56] is an ideal compiler framework for developing a backend for STARPro.

LLVM is a generic framework that nicely separates typical compilation process into three layers. The frontend at the top level can be developed to read a language as input, then passed onto the middle layer that represents the program in a generic intermediate format. This is where optimizations developed for the middle layer can be applied, then the optimized graph can be passed to the bottom layer that calls the backend to generate low level code.

The concept of developing a STARPro backend for LLVM can be brought further in the long term. A frontend to Esterel can be developed for LLVM so as to fully control the
compilation process from Esterel and the C host procedure calls to STARPro assembly instructions. This requires integrating the UCCFG\textsubscript{sd} into LLVM’s intermediate format. If completed, Esterel code can also benefit from the optimizations performed at the middle layer.

### 7.1.2 Fine-tuning compilation for multi-core

There are several factors contributing to the speedups gained from executing Esterel on a multi-core processor compared to sequential execution. Among those factors is the question of load balancing. Chapter 6 has dealt with load distribution using both static and dynamic approach, with dynamic load distribution out-performing the static approach. Ideally, dynamic load distribution would work to its full potential if context switching is overhead free. In reality, the context switching overhead reduces the effectiveness of this approach. As a compromise, instead of distributing every individual thread, some threads are grouped together to reduce context switching.

Currently, a thread can only be forked up to two times in the thread hierarchy before child threads are grouped together. This is where the algorithm for static load distribution can be put to use by analyzing the work load of each thread. For example, if the load reaches some threshold, then a child thread may be mapped to a different core than its parent thread, instead of remaining together. However, to be even more precise than the current approximation technique, static timing analysis [57, 58] would be required.

### 7.1.3 Handling larger class of Esterel programs

One of the most challenging aspects of compiling Esterel is determining the causality of the program. Causality issues revolve around the question of handling cyclic dependencies in Esterel. Such cyclic programs are not always non-causal, as some of the dependency cycles can be broken [59] if the control-flow of the program does not allow such communication to take place.

The dynamic signal resolution algorithm presented in Chapter 6 was originally developed to resolve signals at run-time for multi-core execution. The algorithm’s ability to remove false dependencies plays a key part in the quest to compile a causal cyclic programs. However, not all cycles can be easily broken at compile time. In some cases, the cycle can only be broken at run-time. This is where run-time signal resolution comes into play. This would, however, require the causality of the program to be checked thoroughly before it can be executed. Failing to reject non-causal programs would result in programs deadlocking at run-time.

A possible solution is to cast the causality problem into a Boolean satisfiability issue, and solve the problem using a Boolean satisfiability solver. The locks can be represented
as Boolean bits. When a signal lock consisting of these bits is released, all bits representing the lock from all potential emitters would be set to either zero or one, depending on the actual implementation. Furthermore, signal communication in Esterel can readily be represented using Boolean equations. When combined with signal locks, the complete program can be transformed into Boolean equations and passed into a Boolean satisfiability solver to check whether all locks are always released. Satisfying this condition would confirm the program being causal, otherwise the program would be rejected.

7.2 Final remarks

The compilation techniques of Esterel presented in this thesis offer better performance for different needs. An Esterel programmer has the choice of generating code for a processor optimized for executing Esterel, or parallel execution of Esterel using the compiler designed to support multi-core.

For small controller-oriented embedded systems, the proposed processor architecture coupled with its tailored Esterel compiler produces efficient code while consuming significantly less memory footprint than conventional Esterel compilers. Large concurrent Esterel programs with intensive data computation are very well suited for distributed execution using the multi-core version of the Esterel compiler. The choice offered by these prototype tools cover the typical requirements of developing embedded software using Esterel.
A

List of Publications

The following is a list of research articles that are based on the techniques presented in this thesis:


References


