Compiling Esterel for Multi-Core Execution

Wahbi Haribi
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
wah@informatik.uni-kiel.de

ABSTRACT

The aim of this paper is to present an approach to enable Esterel programs to be effectively executed on multi-core processors. Esterel is a synchronous programming language used for programming reactive systems. The advantage of Esterel is that it allows the simple expression of parallelism and preemption. Esterel programs can easily consist of a large number of threads that the operating system (OS) cannot manage efficiently. Therefore they are generally compiled into sequential code. However, the sequential code can only run on single-core processor. The work by Simon Yuan, Li Hsien Young and Partha S. Roop [10] presented in this paper shows how Esterel threads can be distributed in a balanced way on different processors. Experimental results are also presented to evaluate the practicality and usefulness of this approach.

Keywords

Compiling Esterel, distributed programming, distributed Esterel

1. INTRODUCTION

Esterel is a synchronous programming language [4] for reactive systems. It continuously reacts to events coming from the environment. A reaction is said to take no time. Events occurring during such a reaction are considered simultaneous. To describe reactive systems we need a programming language that allows the simple expression of deterministic timing predictable concurrency and preemption [1] and Esterel fits that.

Concurrency is the basic model for multithreading [8]. Two threads are called *concurrent*, if they can operate independently. The scheduler is free to choose the order of execution. Multithreading generally occurs by time-division on a single processor. The processor switches frequently enough between different threads so that the illusion of parallelism is achieved and threads seem to run at the same time.

Esterel threads run in lock-step according to a global clock and Esterel programs can easily consist of a large number of threads that the OS cannot efficiently manage. Therefore they are generally compiled into sequential code [7]. This practice does not take advantage of multi-core processors. On a multi-core processor, the threads can really run at the same time. Threads are distributed among different cores. The improvement in performance gained by the use of a multi-core processor depends on the way the software algorithms are implemented and used.

The approach used by Yuan et al. [10] to overcome the limitation of the use of multi-core processor is to allow the Esterel threads to run in parallel on the different available cores. Threads can be distributed in a balanced way with the aid of a heuristic algorithm. So the threads then may be dynamically scheduled. Because threads communicate together, they have to be synchronized at the boundary of each clock cycle and within a clock cycle.

Experimental results derived by Yuan et al. are also presented in this paper to show the efficacy of the proposed approach on the multi-core processors.

The rest of this paper is organized as follows. Section 2 outlines the Esterel Language, presents an Esterel example program and describes how to compile Esterel programs. Section 3 introduces the Concurrent Control-Flow graph. Section 4 presents the approach used to dynamically schedule Esterel programs. Section 5 presents the experimental results. Finally, further steps are discussed in the Conclusion.

2. ESTEREL

Communication and computation in Esterel happen instantaneously, unless threads pause explicitly. Events occurring at the same clock cycle (*tick*) are considered simultaneous. The program reads in each clock cycle the inputs, computes its reaction and suspends. The communication is done through signals. Each signal is only present or absent in a *tick*, never both and each signal is present in a *tick* only if it is emitted in this *tick*. Esterel has eleven primitive statements.

2.1 Esterel primitive statements

Signals are emitted using the statement "emit S" and tested using the statement "present S then P else Q end". If S is present, P is executed and Q otherwise. Signals are global or local to a block. The "signal S in P" statement introduces the local signal S to the scope of statement P. Statements execute and terminate in zero time or delay for a prescribed number of cycles. They can be built sequentially in a semicolon-delimited sequence (P ; Q). When P terminates, Q is executed. Statements delimited by double vertical bars (P || Q) are executed concurrently. Signals between P and Q are transmitted instantaneously. The parallel statement (P || Q) terminates when P and Q terminate. A loop is

written "loop S end" with S a non instantaneous statement. With the statement "pause", the Esterel program waits for the next cycle. The statement "suspend P when S" runs P in cycles where S is absent. The statement "trap T in P end" runs P until a "exit T" statement is executed or P terminates itself. The "**nothing**" statement does nothing and terminates instantly.

2.2 An Esterel example

An Esterel example is introduced in Listing 1. It combines delay, instantaneous signal reaction, preemption and concurrency. The signals R and S are the input signals from the environment. The signal O is the output to the environment. The program starts by waiting for input S (line 4). Two local signals A and B are introduced (line 5). The program is divided afterwards into two concurrent threads (lines 7-13 and lines 15-21) surrounded by an *abort* statement (lines 6 and 22). If R is present at any time after the initial tick, the internal statement will be preempted and the program will terminate. The first thread (lines 7-13) emits A every tick and checks the status of signal B (lines 9-11). If B is present, O will be emitted (line 10). The second thread (lines 15-21) emits B (line 19) in alternate cycles when A is present.

We notice in this example the bidirectional communication of signals A and B between the first and the second thread.

Listing 1: An example depicting a simple Esterel module.

```
1  module Example:
2  input R, S;
3  output O;
4  await S;
5  signal A, B in
6    abort
7      loop
8        emit A;
9        present B then
10         emit O
11       end present;
12       pause
13     end loop
14     ||
15     loop
16       pause;
17       pause;
18       present A then
19         emit B
20       end present
21     end loop
22   when R
23 end signal
24 end module
```

Esterel programs are translated into software or hardware. The following subsection describes how Esterel programs are translated and introduces some Esterel compilers.

2.3 Compiling Esterel

There are three main techniques to compile Esterel: translation to state machines, translation into circuit, and software generation [7]. Compilers differ in structure and efficiency of the code they generate:

- Automata-based Compilation produces a state machine. It simulates an Esterel program in every possible state and generates code for each one. The generated code is fast. However it is not practical for large programs.
- The Netlist-based compiler translates programs into Boolean logic circuits. It scales well and it is good for causality problems. However the code is slow and inefficient.
- The EC compiler of Edwards [3] generates a control-flow code. Esterel has a fairly natural translation into a concurrent control-flow graph (CFG). The Control-flow approach scales well as the Netlist-based compiler. It generates a fast code. However it is bad at causality.

Recently an approach to compile Esterel language was defined by Dumitru Potop-Butucaru and Robert de Simone [6]. This approach uses a *GRaph Code (GRC)* intermediate representation to achieve significant performance and generality improvements. The GRC approach uses a *hierarchical state representation (HSR)* and a *concurrent control-flow graph (CCFG)* to preserve most of the structural information of the Esterel program. The *HSR* [2] represents the modular structure and an abstraction of the syntax tree of an Esterel program. However the *CCFG* represents the computation of reactions. It is used as an intermediate representation and uses a static scheduling for generating wellstructured code. The GRC is used as an intermediate representation for Esterel in this approach.

The section below introduces the CCFG of the GRC.

3. THE CONTROL-FLOW GRAPH

The CCFG [6] describes explicitly the concurrent control-flow of Esterel through nodes:

- Action nodes (rectangles) represent signal emissions.
- Enter nodes (ellipses) define state encoding.
- Test nodes (diamonds) represent signal tests.
- Switch nodes (double-diamonds) represent states selections.
- Fork (triangles) and join (inverted triangles) nodes represent concurrency.
- Terminate nodes (hexagon) reflect the completion code.

Figure 1 depicts the CCFG of the Esterel program introduced in Listing 1. The program starts by testing the enter node *state*0. The enter node *state*0 has initially the value 3. *State*0 is then set to 2 and the program pauses for one tick. The "pause" statement is represented by the terminate node having completion code 1. It is not surprising to see that because the "await S" statement can not be solved in the first tick. The *state*0 is then switched and a test node checks if S is present. The program waits if S is absent.

When S is present, local signals A and B are set to absent. $State0$ is set to 1 and the concurrent threads are entered by the fork node. The first thread, located on the left side, emits A and tests the status of B. The output signal O is emitted if B is present. The program pauses for one tick before it switches to the $state0 = 1$ branch. The local signals A and B are then set to absent. The status of R is tested. If R is present, $state0$ is switched to 0 and the program terminates. If R is absent, the fork node is entered. The thread, located on the left side, emits A and tests the status of B. O is emitted if B is present. The thread on the right side emits B if $state1 = 0$ and A is present or switches $state1$ to 0 otherwise. The $state0 = 1$ branch is repeated until the signal R is present. The dashed arrows indicate the signal dependencies between A and B.

The nodes $A!$, $A?$, $B!$, $B?$ and ∞ are introduced in the CCFG to solve the signal dependencies between A and B. They are described in the following section.

4. PARALLELIZATION OF ESTEREL

Enabling Esterel programs to be effectively executed on multi-core processors [10] requires two global steps. The first step is the extension of the GRC to achieve run-time signal resolution. The second step is the code partitioning to distribute the code of the parallel branches of different processor cores.

The following subsections introduce the required steps.

4.1 Run-time signal resolution

The solution, described by Yuan et al. [10], to achieve run-time signal resolution is to use *semaphores* to lock shared signals. The value of the semaphore of a shared signal is set to the maximum number of potential emitters for that signal. This initialization is done at the start of each tick. The presence of the signal is established when the semaphore reaches zero. The semaphore is set to zero if the associated signal is emitted. The semaphore is decremented, if a potential emitter for this signal is ruled out in the control-flow during execution.

Threads will wait for the semaphore to reach zero before testing the status of an associated protected signal. A cyclic executive allows other threads to be dynamically scheduled for execution if a thread blocks. Proceeding to the next tick is allowed when all threads of all cores have been resolved.

Figure 1 shows how to implement the checking and decrement of the semaphores on the CCFG. To protect the signal A and B with counting semaphores, two *guard* ($A?$ and $B?$) nodes are inserted immediately before the test node of each signal. To reset or decrement a counting semaphore, *resolution* nodes are inserted. The *resolution* nodes $A!$ and $B!$ inserted respectively below the emission of A and B resets the counting semaphore. However the *resolution* node ($B!$) inserted under the absent branch of A and the one inserted under the $state1 = 1$ branch decrement the counting semaphore of B. The counting semaphore of A and B will be initialized with 1 because A and B have only one emitter each one. If A is emitted during a tick, the counting semaphore of A is reset and the signal A is unlocked having a present status. If this occurs in the third tick, the second thread detects immediately the presence of A and emits B and the first thread emits the output signal O due to the bidirectional communication.

Listing 2: The unlocking algorithm [9].

```
1  SRN is short for Signal Resolution Node
2  procedure find_last_emitter
3    foreach thread t having data predecessors do
4      foreach emitted signal s do
5        firstNode := t.first
6        lastNode := t.last
7        insert SRN at the start of t
8        get_insertion_points(firstNode, s)
9      end
10   end
11 end
12 M denotes a stack of conditional nodes
13 procedure get_insertion_points(n, s)
14   if n has been visited
15         or n = lastNode then
16     return
17   end
18   add n to the visited set
19   if n = fork then
20     n := matching join node of fork
21     add n to the visited set
22   end
23   if n is an emitter node and emitted signal is s then
24     if previous SRN r is above n then
25       remove r
26       if n is under a conditional node c then
27         mark current branch as having a SRN
28       end
29     end
30     insert a new SRN after n
31   else if n has > 1 successors then
32     insert n in M
33   end
34   children := number of successors of n
35   foreach successor suc of n do
36     if children > 1 then
37       if n is under a conditional node c then
38         if visited branches of c contain SRN then
39           insert SRN at start of current branch
40         end
41       end
42     end
43     get_insertion_points(suc, s)
44   end
45   if children > 1 then
46     if any successor branch contained SRN then
47       insert SRN at start of sibling branches having
48       none
49     end
50     remove n from M
51     if n is under a conditional node c then
52       mark branch containing n as having SRN
53     end
54   end
55 end
```

The *infinity* completion code, introduced in Figure 1, blocks threads from proceeding. In this case, the scheduler contin-

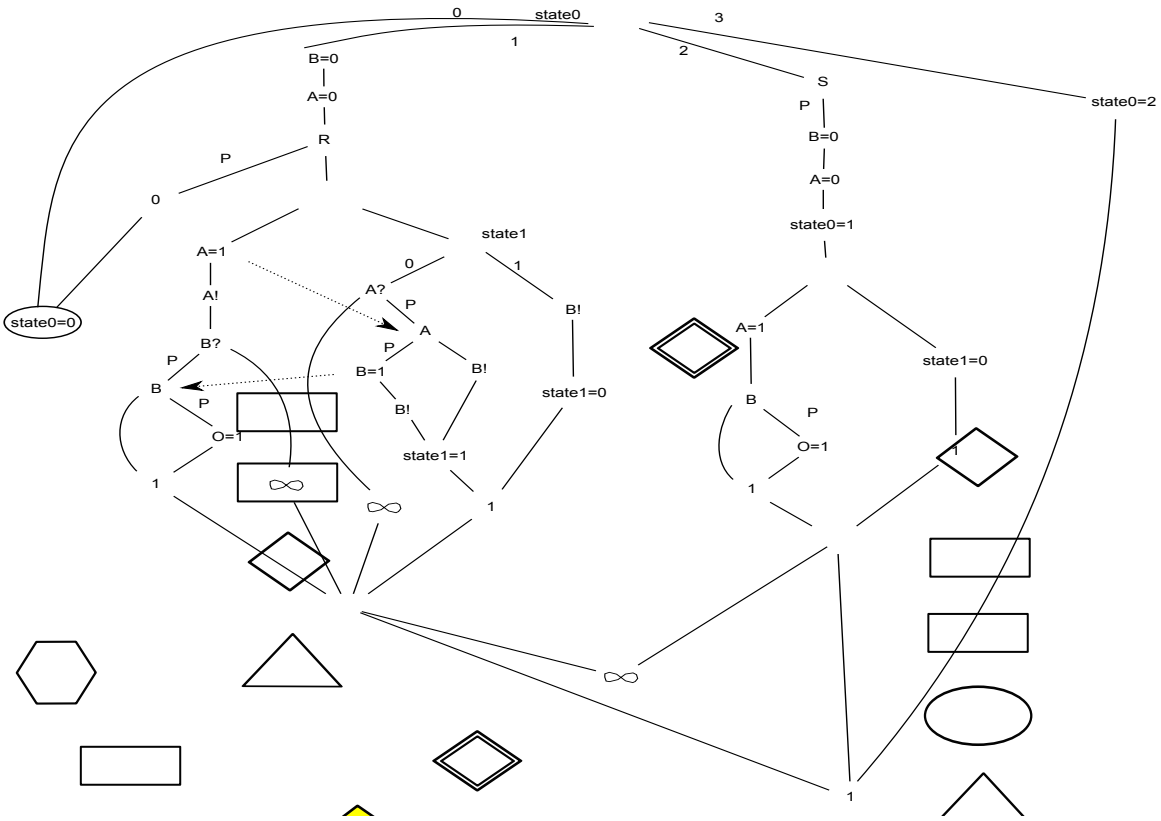

Figure 1: This figure shows the corresponding concurrent control-flow graph of the Listing 1.

ues scheduling the blocked thread until the blocked signals are resolved.

Yuan et al. [10] describe a two part approach to make runtime signal resolution work and to find the correct points to insert the resolution nodes.

First Part. The first part is to block the testing of a signal until its status is known. That is done by inserting the signal guards.

Second Part. The second part is to unlock signals if they are emitted or when they can no longer be emitted by inserting the resolution nodes.

The algorithm introduced in Listing 2 finds the earliest points to insert the resolution nodes.

The procedure find_last_emitter inserts a signal resolution node (SRN) at the start of threads containing signal emitters that lead to data dependencies. Then it calls the procedure get_insertion_points for each found thread.

The procedure get_insertion_points relocates the inserted SRN to the right place, if it is necessary. If the current visited node n is a fork, the nodes enclosed by n are skipped (lines 18-22). If the current visited node n is the data predecessor of a testing node, the SRN inserted earlier is moved after n (Lines 23-30). If the parent node of n is a conditional node, the current branch is marked as having a SRN (lines 26-28). The algorithm checks (lines 35-44) if the current visited node n is under a conditional node and if the visited branches of this conditional node contains a SRN. If that is the case, a SRN is inserted at the start of the current branch. Hereafter, the procedure get_insertion_points is called for the successor node of n (line 43).

The rest of the algorithm (lines 45-54) checks if any successor branch contains a SRN. If that is true, a SRN is inserted at start of sibling branches which no one contains. The current node n is then removed from the stack of conditional nodes. Finally, the branch of n is marked as having a SRN, if n is under a conditional node.

4.2 Code partition

The algorithm in Listing 3 partitions the program for execution on a multi-core system. It ensures that the transition between nodes in a thread always takes place in the same partition. Each partition is assigned to a processor core. The number of generated partitions depends on the number

of the offered processor cores by the executing platform.

The procedure forward_dfs searches the top-level fork through the CCFG (line 8) and estimates the cost of each branch under it.

If the current visited node n is the top-level fork, n is affected to r (line 9) to keep it tracked and the algorithm estimates the load of all branches under r (lines 10-14). The cost of the current branch being traversed is denoted as c and calculated only when traversing under a branch of a fork node (lines 21-23). c is calculated and inserted into the vector of costs C. C is then sorted in ascending order and the procedure load_balance(C) is called.

This procedure load_balance assigns each branch from the vector of costs C to the processor with the least load in P (lines 31-37).

Listing 3: The distribution algorithm [9].

```
1  r denotes the top level fork in the root thread
2  c denotes the cost of the current branch being traversed
3  c_n denotes the cost of the current node being visited
4  C denotes a vector of costs associated with each branch
       under a fork
5  procedure forward_dfs(n)
6    if n is visited then return
7    add n to the visited set
8    if n = fork and r = 0 then
9      r = n
10     foreach control successor s of n do
11       c := 0
12       push c into C
13       forward_dfs(s)
14     end
15     sort C in ascending order
16     load_balance(C)
17     r := 0
18     empty C
19     return
20   end
21   if C ≠ ∅ then
22     c := c + c_n
23   end
24   foreach control successor s of n do
25     forward_dfs(s)
26   end
27 end
28
29 p denotes the load of the processor
30 P denotes a vector of loads associated with each processor
31 procedure load_balance(C)
32   foreach branch cost c ∈ C do
33     sort p ∈ P in ascending order
34     assign c to the first processor p ∈ P
35     p := p + c
36   end
37 end
```

5. EXPERIMENTAL RESULTS

This approach is evaluated [9] with programs that vary in line count, number of threads and the class of the Esterel program (control-dominated or data-dominated). The Table 1 shows the variation of the different programs.

Name	LOC(Esterel)	LOC(C)	Threads	Class
ABIC	21	0	3	C
WW (Estbench)	1087	676	22	C-D
WW09	317	0	7	C
LiftController	272	230	12	C-D
LZSS	42	462	5	D

Table 1: The selected programs [9]

The ABIC program is control-dominated and has the shortest code size. It contains 21 lines of code. The WW (Estbench) program is both control-dominated and data-dominated and has the largest code size. It contains 1087 lines. The WW09 contains only control-dominated code. The LiftController program is both control-dominated and data-dominated. The LZSS program has a parameter, witch varies the amount of data computation.

Example	1 core	2 cores
ABIC	3.5ms	4ms
WW (Estbench)	357ms	410ms
WW09	349us	347us
LiftController	2.57us	3.06us

Table 2: Performance comparison on PC [9]

The programs are tested on a 2.4GHz Intel Core 2 Quad desktop PC with a Linux OS and a dual-core Xilinx Microblaze system[1] without an OS.

The experimental results presented in Table 2 was performed on the desktop PC and reveal that short and control-dominated programs do not benefit from the Parallelization approach.

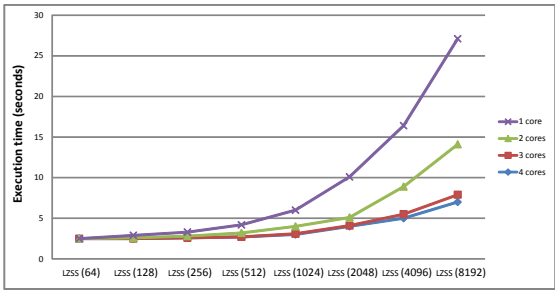

Figure 2: Performance comparison of the LZSS program on PC [9]

Figure 2 depicts the experimental results generated by the LZSS program on the desktop PC. The speedup of the program increases as the data-computation of the program increases. The program speedup with four cores reaches 3.7 times the speed measured on the single core.

The programs were also tested on the Xilinx Microblaze system. The Figure 3 presents the comparison of the average

[1]A microprocessor core that can be implemented using logic synthesis.

case reaction time (ACRT) of the programs on a single core with the ACRT on a dual-core (labeled on the Figure with DMB). The ACRT is the sum of the reaction times to complete an input trace divided by the number of *ticks*. The results show that ABIC and WW(Estbench) performances become worse. The WW09 and LiftController programs are a little bit better.

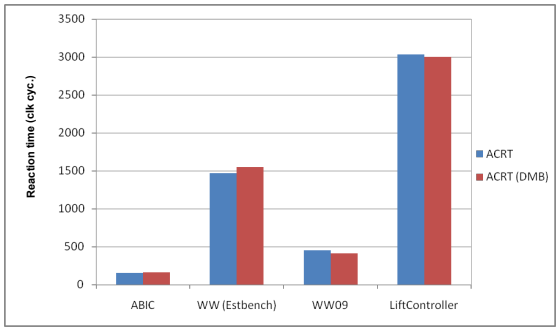

Figure 3: Performance comparison on dual-core Microblaze (lower better) [9]

The LZSS speedup on the Xilinx Microblaze system remains nearly the same as the speedup mesured on the desktop PC.

Examples	Max(%)	Min(%)	Average(%)
ABIC	52	48	50
WW(Estbench)	72	1	40
WW09	49	9	34
LiftController	96	94	94
LZSS	0	0	0

Table 3: Comparison of processor idle time on the dual-core Microblaze in % [9]

The processor idle time between each fork-node pair is also measured on the OS independent platform to evaluate the load balance. Table 3 presents the maximum, minimum and average processor idle time measured between each fork and its related join. The most balanced program is the data-dominated one. The LZSS is the most balanced program. It has a 0% processor idle time on average. The worst balanced one is the LiftController program witch has a 94% processor idle time on average.

These experimental results illustrate that performance gain depends on the amount of dependency between threads on different cores, the size of the program and the effect of such distribution on the processor idle time.

6. CONCLUSION

To enable Esterel programs be effectively executed on multi-core processors, two global steps are required: extension of the GRC to achieve run-time signal resolution and code partition to distribute the code of the parallel branches of different processor cores [10]. Achieving run-time signal resolution is done by using semaphores to lock shared signals. The testing of signals is blocked until its status is known. Shared signals are unlocked if they are emitted or when they can no longer be emitted. An algorithm ensures that threads are automatically partitioned to different cores.

Experimental results show the efficacy of the proposed approach on the multi-core processors. However it is possible to improve this approach to reach a consistent speedup. Yuan et al. [10] proposed a way to improve this approach which consists of a better load-balancing algorithm based on static timing analysis of the different threads that minimizes the processor idle time.

Recently an approach to generate an efficient concurrent multi-threaded code was defined by Jose et al. [5]. This approach attempts generating multi-threaded code without changing the compiler. The generated sequential C-code is used to generate a process-oriented multi-threaded code witch is more suitable for current multi-core architectures. The multi-threaded code is merged manually. However it is possible to implement its automation.

7. REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE, vol. 91, no. 1*, pages 64–83, 2003.

[2] R. de Simone and D. Potop-Butucaru. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on*, pages 227–236, 2003.

[3] S. A. Edwards. An Esterel compiler for large control-dominated systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 169–183, 2002.

[4] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[5] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Generating multi-threaded code from polychronous specifications. In *Journal Electronic Notes in Theoretical Computer Science (ENTCS) Volume 238 Issue 1*, pages 57–69, 2009.

[6] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Co-Design*, pages 227–236, 2003.

[7] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[8] P. Scholz. *Softwareentwicklung eingebetteter Systeme*. Springer, 2007.

[9] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian. Compiling Esterel for distributed execution. In *proc. Synchronous Languages, Applications and Programming 2006, European Joint Conference on Theory and Practice of Software, March 25- April 2*, 2006.

[10] S. Yuan, L. H. Yoong, and P. S. Roop. Compiling Esterel for multi-core execution. In *14th Euromicro Conference on Digital System Design (DSD)*, pages 727–735, 2011.