Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 37–55
www.elsevier.com/locate/entcs

STARPro — A new multithreaded direct execution platform for Esterel

Simon Yuan [1] Sidharta Andalam [2] Li Hsien Yoong [3]
Partha S. Roop [4] Zoran Salcic [5]

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

Abstract

Esterel programs have traditionally been compiled to software code for general purpose processors or to hardware netlists. This paper, instead, proposes a reactive processor for the direct execution of Esterel. This intermediate approach offers the same flexibility as software compilation, while at the same time, providing much better code size and execution time. The proposed architecture, called STARPro, is a pipelined, multithreaded, reactive processor that provides native support for the direct execution of Esterel. STARPro manages Esterel threads and their scheduling, and also features a hardware preemption unit to assist the handling of the abort constructs in Esterel. In addition to the proposed architecture, we have also developed a new intermediate format called UCCFG$_{sd}$ (unrolled concurrent control-flow graph with surface and depth) to represent the structure of an Esterel program in our compiler. UCCFG$_{sd}$ closely resembles the Esterel source, and it has also been designed with Esterel hardware support in mind, allowing a straight forward translation into STARPro assembly instructions. We have compared the performance of STARPro against a recent reactive architecture and found an average of 37% speed-up in worst-case reaction times, and 38% in average-case reaction times.

Key words: Compilation, concurrency, Esterel, reactive processors, synchronous

[1] Email: iyua002@ec.auckland.ac.nz
[2] Email: sand080@ec.auckland.ac.nz
[3] Email: lyoo002@ec.auckland.ac.nz
[4] Email: p.roop@auckland.ac.nz
[5] Email: z.salcic@auckland.ac.nz

doi:10.1016/j.entcs.2008.01.005

1 Introduction

Esterel is a synchronous reactive programming language [3]. Esterel has formal semantics that guarantee *reactivity* and *determinism* [2], resulting in predictable runtime behaviour that greatly simplies the formal verification of programs. These properties are highly desirable in the design and validation of a special class of embedded systems called *reactive systems* [9].

Reactive systems typically consists of tasks running in parallel. This concurrency is traditionally supported by running an operating system (OS) that manages these tasks. The dynamic nature of an OS makes the programs running in it difficult to debug and verify. The synchronous paradigm that Esterel follows, in contrast, abstracts away the physical time, and synchronizes the program with a global logical clock. All concurrent components of an Esterel program execute in lock-step, evolving in discrete instants of time, known as a *tick*. Execution is assumed to be infinitely fast between *ticks*. Hence, the communication between the concurrent components are 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. We illustrate some aspects of this complexity using the example shown in Fig. 1(a).

The `demoloop` example demonstrates three key features of the language: concurrency, synchronous preemption, and instantaneous broadcast communication. It consists of a single module, with its input and output interface signals declared on lines 2 and 3 respectively.

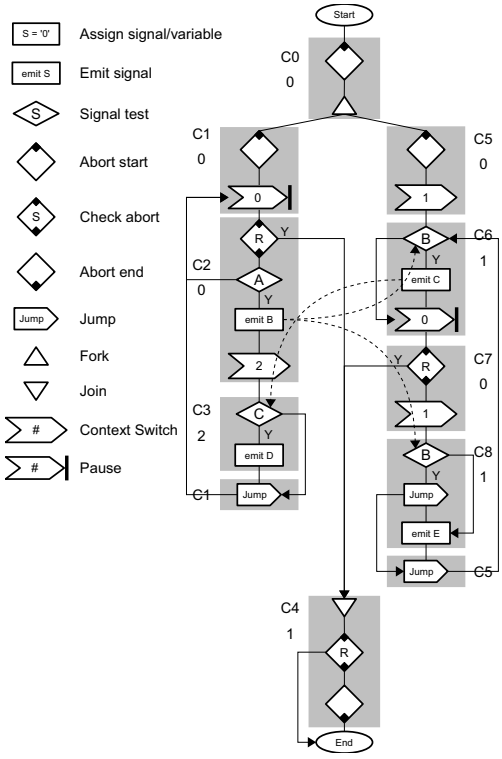
The program consists of two parallel threads, beginning from lines 5–11, and lines 13–21 respectively. Both threads are enclosed by their respective non-terminating loops. The '||' operator is used to denote their synchronous concurrency. The first thread begins by awaiting for input `A` (line 6). The `await` statement always pauses for the first instant, and will continue to do so in subsequent instants until its delay predicate (`A` in this example) becomes true. Once `A` becomes present, `B` will be emitted.

The emission of `B` is broadcast to the second thread, which instantaneously reacts to it by emitting `C` (lines 14–16). The emission of `C`, in turn, provokes the instantaneous emission of `D` back in the first thread (lines 8–10). Meanwhile, in the second thread, the `pause` statement on line 17 marks the end of its *tick*, and implies synchronization with the first thread before the start of the next instant. The possibility of instantaneous dialog, as well as this implicit synchronization at each instant between concurrent components, are two factors that make the efficient software implementation of Esterel challenging.

In the subsequent instant, the first thread will again wait for another occurrence of `A`. If `A` is not present this time, the latter thread will respond by emitting `E` (lines 18–20). The ability to react instantaneously to both signal

```
1  module demoloop :
2  input A, R ;
3  output B, C, D, E ;
4  abort
5       loop
6            await A ;
7            emit B ;
8            present C then
9                 emit D
10           end
11      end loop
12       ||
13      loop
14           present B then
15                emit C
16           end ;
17           pause;
18           present B else
19                emit E
20           end
21      end loop
22 when R
23 end module
```

(a)



(b)

Fig. 1. The `demoloop` example: (a) Esterel source (b) Unrolled Concurrent Control-Flow Graph (UCCFG$_{sd}$)

presence and absence is a crucial issue, making the scheduling of programs in Esterel non-trivial.

Meanwhile, should the signal `R` become present at any time after the first instant, both the parallel threads will be preempted at precisely the same instant by the `abort` construct enclosing them on lines 4–22. This will instantaneously terminate the program. This distinct behaviour for the start and resumption instants are referred to as the *surface* and *depth* behaviours respectively [15].

Several approaches exist for dealing with these complexities in the compilation and execution of Esterel programs. These include hardware compilation [2], software compilation for general-purpose microprocessors [5,15,8], and architecture-specific compilation for reactive processors optimized for Esterel [6,10]. 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 artificial synchronization mechanisms to be added to preserve Esterel's semantics. Such mechanisms introduce extra execution overhead and increase the required memory footprint.

The architecture-specific approach relies on custom microprocessors that have been augmented with an instruction set, which enables the efficient mapping of Esterel statements to assembly code. This approach yields very compact software code, as well as efficient execution, and will be the focus of this paper. We present a novel multithreaded processor, named STARPro (Simultaneous multiThreaded Auckland Reactive Processor), and an Esterel compiler for it, that achieves significant speed-up and code size compaction over traditional methods for software implementations of Esterel.

Multithreading in our processor design differs to conventional multithreaded processor in the sense that threads are interleaved rather than executing in parallel. The hardware provides the facility to store and context switching between threads.

The rest of this paper is organised as follows. Section 2 reviews previous work related to architecture-specific execution of Esterel. Section 3 then presents STARPro's processor architecture, which is followed by a description of its instruction set architecture (ISA) in Section 4. Section 5 will cover aspects on the code generation from the intermediate format and the execution semantics. In Section 6, we show the experimental results obtained for some benchmarks. We finally end with some concluding remarks in Section 7.

2 Related work

The EMPEROR multiprocessor architecture [6] 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 executed Esterel programs by resolving signal dependencies during run-time using a dual-rail encoding of signals [18]. This approach, while achieveing 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 [12,11,13,10]. The KEP series of processors are custom designed architectures that have evolved with incremental support for executing Esterel. The most recent processor, KEP3a [10], 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 and Esterel execution. At present, KEP3a employs a non-pipelined architecture, which supports Esterel's semantics al-

most entirely in hardware. This approach results in a complex hardware design, with a consequently lower operating clock frequency.

In contrast, this paper presents a novel multithreaded processor, named STARPro, that provides an alterative approach to direct execution compared to KEP3a. STARPro uses variable *tick* lengths and a pipelined architecture to obtain much better average performance compared to KEP3a. This has been achieved using far fewer logic gates for processor implementation, while maintaining code sizes that are comparable to KEP3a for a given Esterel program.

Plummer *et al* [14] have explored another approach of executing Esterel using a virtual machine (VM). A VM provides Esterel supporting intructions for direct execution, similar to the way STARPro works. The key difference is a virtual machine is implemented as software, where STARPro is a hardware platform. Both approaches are superior in code size when compared with traditional Esterel compilers, however the VM approach is significantly slower than traditional Esterel compilers [14] and the VM cannot handle host procedure calls written in for example C.

3 The STARPro Processor Architecture

STARPro's design was based on an existing processor, called REMIC [16]. 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 data path 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 require I/O to be handled synchronously. Hence, we have developed the Esterel Support Unit (ESU) to replace the RFU within REMIC, as illustrated in Fig. 2(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 (see Fig. 2(b)) consists of the Abort Handling Block (AHB) for dealing with preemptions, and the Thread Control Block (TCB) for multithreading support. Unlike most other simultaneous multithreading processors, STARPro does not use separate register files for each thread, but it does, however, provide separate program counters and auxiliary registers for abort handling for each thread. In the following, we will first explain the TCB and AHB, before discussing how the two interact.

3.1 The Thread Control Block (TCB)

The purpose of the TCB is twofold: it is used to *store thread context*, and to *perform thread scheduling*. As depicted in Fig. 2(d), the TCB itself is

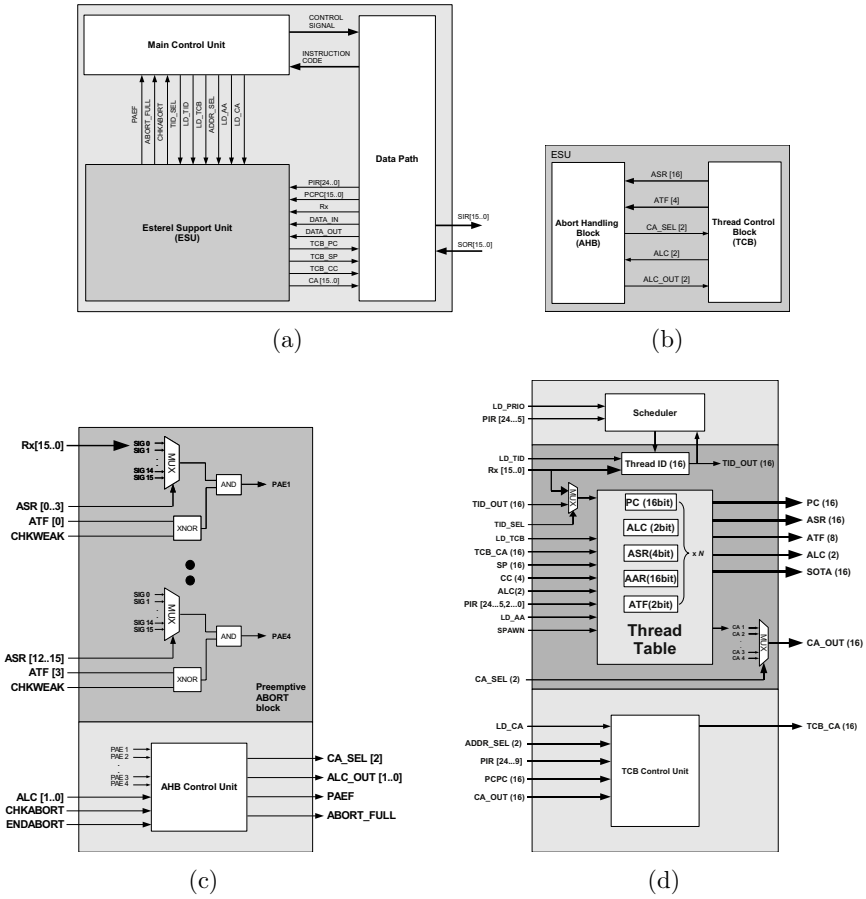


Fig. 2. The STARPro architecture: (a) Overview of hardware blocks; (b) Esterel Support Unit; (c) Abort Handling Block; and (d) Thread Handling Block (TCB)

composed of a scheduler, a thread table, and a TCB control unit.

The thread table stores the current program counter and the abort context [6] 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 in our design, and is limited only to the hardware resources available.

The thread table is indexed by the Thread ID register. The entry indexed by that 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 then become a simple matter of 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

[6] The abort context will be described in Section 3.2.

default value of zero by the STARPro compiler.

The other remaining important component of the TCB is the scheduler. The scheduler stores the priority and a notion of a *local tick* for each thread. We say that the *local tick* for a thread has elapsed whenever a `pause` statement in it is reached. 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 elapse, the global *tick* completes, and a compiler-generated management thread is selected to sample new inputs and to clear all output signals for the next global *tick*.

The distinction between local and global *ticks* is actually the key idea that facilitates the use of variable *tick* durations in STARPro. This idea was first introduced in [6], and has been adapted for our current design. 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 equal to the actual computational time required.

3.2 The Abort Handling Block (AHB)

The AHB is used to monitor aborting signals, and to trigger the appropriate preemptions if 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 than those nested below it. The AHB supports this feature by providing hardware-based priority resolution for the abort constructs. The depth of nested aborts is fully parameterizable in our design. Fig. 2(c) depicts an AHB that has been configured with four levels of aborts for each thread.

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

- `Rx`: This is the bus that connects to a 16-bit register selected from the register file in datapath. The register has to be loaded with the status of I/O signals in a bunch of 16s at a time from memory. It is updated at every *tick*, and is used by the AHB to evaluate the status of the aborting signals.
- `ASR` (Abort Signal Register): This stores the ID of the signal which needs to be monitored during execution of an abort body.
- `AAR` (Abort Address Register): This stores the continuation address, to which the thread must jump, should preemption happens.
- `ATF` (Abort Type Flags): STARPro supports the different types of abortions

in Esterel. Abortions can either be *strong* or *weak*, and may be either *immediate* or *non-immediate*. These are orthogonal to each other, resulting in four distinct behaviours for abortions in Esterel.

- `ALC` (Abort Level Count): Each thread can 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 these abort context of the current running thread to the AHB. The AHB does not contain any memory element and it is purely control. When the AHB detects the presence of the preempting signal, it provides an index (`CA_SEL` in Fig. 2(b)) that selects the continuation address (`AAR`, stored in the TCB), as well as an updated ALC, back to the TCB. The TCB directly provides the continuation address to the datapath, and hence the AAR is the only abort context not passed to the AHB. The activation and deactivation of abort levels are also controlled by the TCB control unit.

The most significant difference between the AHB and the preemption *watchers* in KEP is how the preemption is monitored. STARPro relies on explicit checks at appropriate times using an instruction, where the *watchers* in KEP relies on a physical tick signal in hardware. The correctness of abort semantics of the AHB relies on the compiler at compile time, where the *watchers* rely on the runtime hardware behaviour. The difference in the two approaches results in simpler preemption hardware design for STARPro.

The AHB relies on the control unit to indicate to it when to check for aborting conditions. This is necessary to preserve Esterel's synchronous preemption, and to correctly 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 abortions in the following manner:

- For strong abortions, the AHB starts by evaluating the status of aborting signals, beginning from the outermost to the innermost abort level.
- For weak abortions, the AHB starts by evaluating the status of aborting signals, beginning from the innermost to the outermost abort level.

We describe the reason for this difference. An abort construct in Esterel may contain an abort handler. If an abort handler exists, the handler will be executed when an abortion takes place. A weak abort offers the current executing abort body one last chance to complete the current *tick* before preempting it.

Let us now consider the scenario where a weak abort is nested within another weak abort, and both of them have an associated abort handler. In the instant where the aborting signals for both constructs are present, the program will first execute the inner abort handler up to, but not including, the `pause` statement (if any). Execution will then branch to the outer abort handler.

This chaining of weak abort handlers is the reason behind the different order of checking between the two types of abort constructs. By checking a weak abort beginning at the innermost level, the preemption can be propagated from the inner to the outer levels of aborts.

4 The STARPro Instruction Set Architecture

STARPro uses a 32-bit instruction format. Apart from the common instructions found on a typical RISC processor, we introduce additional Esterel-oriented instructions to support multithreading, signal testing, and preemption. The syntax and description of these instructions are summarized in Table 1.

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 any arbitary arithmetic or logic operation to be performed on signals. STARPro also does not have any dedicated instruction for strong immediate aborts. Instead, this is derived using the `ABORT` instruction, together with the `PRESENT` instruction to test for the aborting condition in the starting instant.

We illustrate the reactive instructions using the example in Fig. 1(a). The equivalent STARPro assembly code for that example is shown in Fig. 3. We

Instruction Syntax	Description
`SPAWN` *Reg StartAddr*	Creates a new thread
`CSWITCH` *Priority*	Context switches to a thread and updates the current thread priority
`PAUSE` *Reg*	Marks the end of a tick and context switch to a thread
`PCHANGE` *Reg Priority*	Changes the priority of a thread
`PRESENT` *Sig Reg ElseAddr*	Checks the presence of a signal
`ABSENT` *Sig Reg ElseAddr*	Checks the absence of a signal
`ABORT` *Sig Addr*	Initializes the AHB for strong abortion
`WABORT` *Sig Addr*	Initializes the AHB for weak abortion
`WIABORT` *Sig Addr*	Initializes the AHB for weak immediate abortion
`CHKABORT` *Reg Type*	Checks for preemption of type *Type* (strong/weak) only
`ENDABORT`	Deactivates the current abort level

Table 1
Esterel-oriented instructions

```
1  ; Signal I/O Addresses
2  INPUTS  EQU $FFFE
3  OUTPUTS EQU $FFFF
4  ; External Inputs
5  A EQU 15
6  R EQU 14
7  ; External Outputs
8  B EQU 15
9  C EQU 14
10 D EQU 13
11 E EQU 12
12 ; Local signals
13 ; Variables
14 JOIN EQU $0001
15     ; Initialize variables
16     STR R0 $JOIN
17     ; Keep input signals in register 6
18     LDR R6 $INPUTS ; external inputs
19 ; --------- Start of program ---------
20     ABORT S14 AA0 ; Abort S14=R
21     LDR R1 #1   ; create
22     SPAWN R1 T1 ; thread 1
23     LDR R2 #2   ; create
24     SPAWN R2 T2 ; thread 2
25     LDR R0 #31   ; special thread for
26     SPAWN R0 GTK ; handling global ticks
27     LDR R0 #$0006 ; set thread 1 and 2
28     STR R0 $JOIN  ; to NOT join
29     PCHANGE R1 #0 ; assign thread 1 priority 0
30     PCHANGE R2 #0 ; assign thread 2 priority 0
31     CSWITCH #255 ; set parent thread inactive
32     CHKABORT R6 STRONG
33     ENDABORT
34 AA0 JMP EN ; end program
35 ; --------- Start of thread 1 ---------
36 T1  ABORT S14 CC1 ; Abort S14=R
37 AA1 PAUSE #0 ; assign thread 1 priority 0
38     CHKABORT R6 STRONG
39     PRESENT S15 R6 AA1 ; present S15=A
40     SBIT R7 R7 #B ; emit B
41     STR R7 $OUTPUTS
```

```
42     CSWITCH #2 ; assign thread 1 priority 2
43     PRESENT S14 R7 BB1 ; present S14=C
44     SBIT R7 R7 #D ; emit D
45     STR R7 $OUTPUTS
46 BB1 JMP AA1
47 CC1 LDR R0 $JOIN ; mark
48     CBIT R0 R0 #1 ; thread 1
49     STR R0 $JOIN ; dead
50     SZ DD1       ; threads join if JOIN == 0
51     JMP EE1
52 DD1 PCHANGE R0 #1 ; activate thread 0 at priority 1
53 EE1 CSWITCH #255 ; set thread 1 inactive
54 ; --------- Start of thread 2 ---------
55 T2  ABORT S14 EE2 ; Abort S14=R
56     CSWITCH #1 ; assign thread 2 priority 1
57 AA2 PRESENT S15 R7 BB2 ; present S15=B
58     SBIT R7 R7 #C ; emit C
59     STR R7 $OUTPUTS
60 BB2 PAUSE #0 ; assign thread 2 priority 0
61     CHKABORT R6 STRONG
62     CSWITCH #1 ; assign thread 2 priority 1
63     PRESENT S15 R7 CC2 ; present S15=B
64     JMP DD2
65 CC2 SBIT R7 R7 #E ; emit E
66     STR R7 $OUTPUTS
67 DD2 JMP AA2
68 EE2 LDR R0 $JOIN ; mark
69     CBIT R0 R0 #2 ; thread 2
70     STR R0 $JOIN ; dead
71     SZ FF2       ; threads join if JOIN == 0
72     JMP GG2
73 FF2 PCHANGE R0 #1 ; activate thread 0 at priority 1
74 GG2 CSWITCH #255 ; set thread 1 inactive
75 ; - Start of special thread for global tick handling -
76 GTK GTICK ; special benchmarking purpose instruction
77     LDR R7 #0       ; clear
78     STR R7 $OUTPUTS ; outputs
79     LDR R6 $INPUTS  ; new snapshot of inputs
80     CSWITCH #255
81     JMP GTK
82 EN END
```

Fig. 3. The `demoloop` example translated to STARPro assembly

start, first, by explaining the reactive instructions used in this program, and defer the discussion on the translation process to Section 5.

Starting with `ABORT` on line 20, the first abort level is configured here to watch for signal 14 (signal `R`). Then, the program forks two concurrent threads. This is accomplished using the `SPAWN` instruction on lines 22 and 24, which initializes thread 1 and 2 to start at label `T1` and `T2` respectively. Line 26 creates the special global tick handler thread. The `PCHANGE` instruction on lines 29 and 30 set the initial priority of thread 1 and 2. Finally, the `CSWITCH` instruction on line 31 completes the thread-forking process by setting the current (in this case, the root) thread inactive. The priority number of 255 is the lowest possible priority (indicating an inactive thread), while 0 is the highest. At this point, either thread can be scheduled as both have the same priority in the starting instant. The scheduler selects whichever it finds first and does a context switch to the selected thread. The `PAUSE` instructions found on several lines across thread 1 and 2 essentially does the same thing as `CSWITCH`, except that the `PAUSE`, in addition, also marks the end of a *local tick* for the currently executing thread.

In order to achieve a simpler hardware design, the abort constructs are kept local to the threads that they have been declared in. When a thread is forked, the aborts within it are duplicated in the child threads, as was done in [6]. Due to this, thread 1 and thread 2 begin with an `ABORT` instruction

on lines 36 and 55 respectively. These two lines do the same initialization as was done on line 20. Inside the abort body, the `CHKABORT` instruction is appropriately inserted at *local tick* boundaries, such as on lines 38 and 61. 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 abortion 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 33) is used to deactivate the current abort level, and it will not be checked again until it is reactivated. The `ENDABORT` marks the end of an abort body, and the instruction following it is simply a branch to the address of the next instruction after the abort construct.

The `PRESENT` instruction, found in many places such as line 39, is functionally equivalent to Esterel's `present` statement. It tests for the presence of a signal. If it 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 instead. It is provided for code compaction, and to avoid unnecessary branching so as to minimize the flushing of the processor's pipeline.

5 Code generation and execution semantics

In order to generate assembly code from the Esterel source, the STARPro compiler uses an intermediate format, called the *unrolled concurrent control-fow graph with surface and depth* (UCCFG$_{sd}$), to represent a given Esterel program. We first present the UCCFG$_{sd}$, and then, describe how assembly code is generated from it.

5.1 Unrolled Concurrent Control-Flow Graph

The UCCFG$_{sd}$ is a variant of the UCCFG intermediate format, which was first introduced in [6]. However, the UCCFG is not capable of fully preserving Esterel's semantics, especially for statements that have distinct start and resumption behaviours (also known as *surface* and *depth* behaviours), like that of the `await` statement described in the example of Fig. 1(a). Some statements, like `emit`, are logically instantaneous, while others, like the `await` statement, consumes time (*ticks*). Such non-instantaneous statements have distinct surface and depth behaviours.

To overcome this, we have modified the original UCCFG format, and extended it to explicitly capture both the surface and depth behaviour of every statement in Esterel. This approach adapts the technique used in [15], where the start and resumption behaviours are differentiated using distinct *surface code* and *depth code*.

In [15], 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*.

In UCCFG$_{sd}$, tick boundaries are marked by *pause* nodes, denoted as an arrow with a black bar on the right, as depicted in Fig. 1(b). Using these *pause* nodes, the loop required to execute the CFG of [15] can be completely unrolled. Hence, instead of using a switch statement to select between the surface and depth code as done in [15], code for STARPro can be conveniently represented in this form:

$$surface(\texttt{code});depth(\texttt{code})$$

Using this approach, Esterel statements can be mapped to UCCFG$_{sd}$ nodes rather intuitively. The mapping of the `abort` statement, however, would merit further elaboration. This is actually done in two stages: first, by marking the start and end of the body, and subsequently, by placing the *check abort* node at the desired points. 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, we use the following general rules:

- 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. A *present* node is simply added before the *abort* node to test for the aborting condition.
- The *non-immediate* version of a weak abort also has the *check abort* nodes inserted before the *pause* node of the first instant. The reason for this is descibed below.

The handling of a non-immediate weak abort is subtle when its abort body contains a loop. The first pass through the loop is different from all subsequent passes, as the *surface* part of the loop body gets folded back into the *depth* after the first pass. 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 AHB has been designed to ignore the first `CHKABORT` instruction encountered for weak non-immediate aborts using an additional status bit.

The `demoloop` example contains a strong abortion. In Fig. 1(b), this is indicated through the *start abort* node. Within the abort body, a *check abort* node is placed after each *pause* node in the two forked threads. An *end abort*

node is placed at the end of the abort body. The start and end abort pair, thus, defines the scope of the abort in the graph. The two sibling threads in Fig. 1(b) presented here do not end with *end abort* nodes. These two threads will never reach the end of the abort body due to the loops. For this same reason, the two threads will only join should the abort take place via the *check abort* nodes. The last *check abort* node below the *join* node will only have an effect if the abort in the root thread has an abort handler, which is not the case in this example.

Following the preliminary construction of the UCCFG$_{sd}$, the nodes are clustered into distinct sets to facilitate their static scheduling. This is similar to that done in the CEC compiler [8]. However, unlike CEC, our scheduling is done in hardware using a priority instruction, similar to [10].

Each *pause* node marks the end of a cluster. Additionally, nodes may also be separately clustered due to data depency arcs, as can be seen from the `demoloop` example of Fig. 1(b). A *context switch* node is inserted as such points. The clusters are then assigned priorities based on the depth of the dependency chain. In the case of cluster C3 in thread 1, the depth of the dependency chain is two (C2 to C6, C6 to C3), and hence, it is assigned a priority of two.

The starting clusters of the two forked threads in the `demoloop` example have the same priority, and thus it makes no difference in terms of program behaviour which cluster to be scheduled for the first instant. If C1 is to be executed first, the *pause* node in C1 marks the end of the *local tick* for thread 1, and it will no longer be scheduled until all active threads have completed their *local tick*. Subsequently the *context switch* node in C5 will only cause a context switch to itself in this case. The *context switch* node in C5 is not redundant because the signal producer (C2) can potentially execute in the same instant as the consumer (C6). Thus breaking the first instant of the consumer thread into two clusters (C5 and C6) ensures the potential producer will always be executed prior to the consumer by assigning the consumer cluster with a lower priority.

5.2 Handling schizophrenic programs

Statements in an Esterel program may potentially be executed multiple times within a single *tick*. Such programs are referred to as *schizophrenic* [2,17]. 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* [2]) for each new signal declaration that may potentially occur within the *tick*. This not only complicates the compilation process, but also significantly leads to an 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 to be created. Local signals are

simply implemented as variables in STARPro. Whenever the local signal is declared (redeclared when looping back), 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 ensure 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.

5.3 Code generation

The nodes in the UCCFG$_{sd}$ map very closely to STARPro instructions. Code generation from the UCCFG$_{sd}$ is greatly simplified as there is almost a direct mapping between nodes and assembly instructions. For example, the *context switch* and *pause* nodes directly translate to the `CSWITCH` and `PAUSE` instructions respectively.

The less straightforward ones in Fig. 1(b) are *fork* and *join* nodes. Forking involves the following actions: spawning each child thread, setting the priority and join status (stored as a variable) of each thread, and finally, context switching to one of the child threads and marking the parent thread as inactive. Lines 21 to 31 in Fig. 3 are the translated output for the *fork* node. Joining requires checking the join status, and making sure that all the child threads in the same fork are ready to join before reviving the parent thread. In the `demoloop` example, thread 2 would finish before thread 1. When thread 2 reaches the *join* node, it clears the corresponding bit in the `JOIN` variable, and checks the join status to see if all other sibling threads are ready to join. It then deactivates itself by executing the `CSWITCH` instruction with a priority of 255. These are shown on lines 47 to 53 and 68 to 74 in Fig. 3. When all threads are ready to join (the `JOIN` variable evaluates to zero), the last executing thread of the fork revives the parent thread by changing its priority to a priority lower than the currently executing cluster. When the `CSWITCH` instruction is next executed, the scheduler will select the parent thread.

6 Experimental results

STARPro was synthesized on both CycloneII and Spartan3 FPGA. Its hardware resource usage on Spartan3 is presented in Table 2 for comparison with KEP3a [10]. Since the number of threads supported by STARPro is parameterizable, we synthesised the design for 2 to 512 threads to examine the relationship between the resource usage and the number of threads. Table 2 also shows this relation for KEP3a, when configured for 2 to 120 threads. The table clearly shows that STARPro consumes far less resources than KEP3a for a given number of threads, using an order of magnitude less for the starting

configuration of two threads. The number of I/O ports (memory mapped) configured for STARPro will not significantly change the hardware resource usage as accessing I/O ports is a purely generic memory operation.

STARPro @167MHz	Max. Threads	2	4	8	16	32	64	128	256	512
	Slices	320	391	488	678	851	1479	2710	5251	10137
	Gates(k)	24	24	26	29	52	89	173	342	682

KEP3a @60MHz	Max. Threads	2	10	20	40	60	80	100	120
	Slices	1295	1566	1871	2369	3235	4035	4569	5233
	Gates(k)	295	299	311	328	346	373	389	406

Table 2
Quantitative comparison of harware resource usage

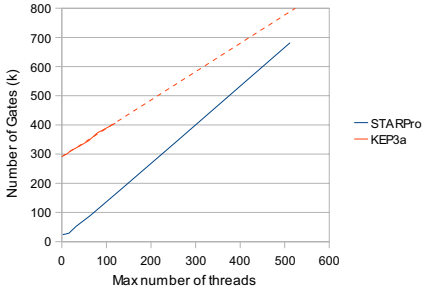


Fig. 4. Comparison of hardware resource usage between KEP3a and STARPro

Fig. 4 portrays the difference in usage of resources between the two processsors. Both processors exhibit a linear relation between the number of logic gates and the level of multithreading support provided. KEP3a, however, incurs a very high initial gate usage at 295k, while STARPro only consumes an initial gate count of 22.9k. This gap narrows as the number of threads increases. Extrapolating this graph reveals that the resource usage of STARPro and KEP3a will meet when the number of threads reach 9000. This means that for an application with more than 9000 threads, the resource usage of STARPro will begin to exceed that of KEP3a. Since most realistic embedded applications are anticipated to use well below 9000 threads, STARPro can be considered to be more efficient in terms of resource usage.

The benchmark programs presented here have been selected from EstBench [7]. All the selected programs are also present in [10] for comparison. The benchmarks were evaluated in three aspects. First, we compare the worst-case and average reaction times for KEP3a and STARPro. The optimized results for KEP3a were taken from [10]. Then, we compare the generated code size. Finally, we show the effects of a pipelined architecture in terms of the speedup obtained.

To evaluate STARPro's compiler, we compared it against four other Esterel compilers, namely CEC v0.4 [8], EEC2 [18], and the V5 [3] and V7 Esterel compilers [4]. These compilers produced C code from the Esterel source, which we compiled for the NIOS-II [1] 32-bit RISC processor. NIOS is a softcore

Module Name	KEP WCRT (clocks)	STARPro WCRT (clocks)	Avg. Speedup	KEP ACRT (clocks)	STARPro ACRT (clocks)	Avg. Speedup
abcd	135	83	1.63	84	42	2
abcdef	201	121	1.66	117	57	2.05
eight_but	267	96	2.78	153	87	1.76
chan_prot	117	140	0.84	54	55	0.98
reactor_ctrl	51	43	1.19	39	39	1
runner	30	88	0.34	6	35	0.17
example	42	46	0.91	24	30	0.8

Table 3
The worst and average case reaction time

Module Name	Code size (bytes)						
	CEC	EEC	V5A	V5	V7	KEP3a	STARPro
abcd	9152	10424	7869	10616	6232	756	824
abcdef	19407	21068	23568	34536	15464	1134	1236
eight_but	3648	4028	3628	6816	6804	1512	1620
chan_prot	4512	6000	6532	10464	9168	297	576
reactor_ctrl	2300	4608	3708	4364	2592	171	248
runner	4524	5312	5684	5560	4148	175	1072
example	3172	3556	3252	3660	2348	139	296

Table 4
Code size comparison using different compilation techniques

processor, provided by Altera as part of its development tools for its CycloneII FPGA. All C programs were compiled using the *nios2-elf-gcc* compiler with level-2 optimization (-O2).

We start by comparing the execution times of the two reactive architectures, KEP3a and STARPro. Execution traces were generated using Esterel Studio's Coverage Analysis tool, which were also used for the benchmarks in [10]. The worst-case and average-case reaction times for KEP3a and STARPro are shown in Table 3. Although KEP3a has almost a one-to-one mapping between Esterel statements and assembly instructions, STARPro is still able to achieve, on average, a 37% speed-up in worst-case reaction time (WCRT), and a 38% speed-up in average-case reaction time (ACRT). The exception where KEP3a significantly excels in performance is the `runner` example. The example involves counting of signal occurrences. In KEP3a, such counting is done in hardware, whereas STARPro relies on software to do this.

The code sizes for the software compilers were obtained from the size of the object files generated by the `nios2-elf-gcc` compiler. The approach taken by KEP3a and STARPro consistently resulted in much more compact code compared to the conventional software approach, as depicted in Table 4. The `runner` example again shows that code sizes are more compact with KEP3a's hardware-oriented approach. STARPro has on average 40% larger code size than KEP3a.

Table 5 shows the performance gain from a pipelined architecture. The clock cycles shown in the table represent the total number of clock cycles required to complete each program with a given execution trace. The same

Module Name	Clocks	Instructions	Clk/inst	Speedup
abcd	21338	10489	2.03	1.47
abcdef	254439	123454	2.06	1.46
eight_but	24645	13439	1.83	1.64
chan_prot	24167	13181	1.83	1.64
reactor_ctrl	544	290	1.88	1.6
runner	1090222	703585	1.55	1.94
example	274	160	1.71	1.75
Average	1415629	864598	1.64	1.83

Table 5
Effectiveness of pipelining in clocks per instruction

applies to the instruction count. Multiplying the instruction count by three, we obtain the total numbr of clock cycles required for a non-pipelined processor. The effect of pipelining results in an average speedup of 1.83.

In summary, execution of Esterel using reactive processors yields much better code size and execution times compared to conventional software approaches. The STARPro architecture proposed here, also provides much better execution times with less hardware resources compared to the latest KEP processor, while suffering only a minimal code size penalty. In general, the STARPro architecture is simpler than KEP3a's, as its instructions are also much simpler. 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. Consider, for example, count delays in Esterel. KEP has direct support for this in its ISA, while STARPro does this in software. Also, for abortions in Esterel, the AHB requires `CHKABORT` instructions to be inserted at appropriate points to emulate Esterel behaviour. This approach leads to a slight code size penalty compared to KEP3a. However, STARPro's simpler hardware not only operates at a higher frequency, but also executes Esterel programs faster, in both the worst and the average cases.

7 Conclusions

We have presented a direct execution platform for Esterel with multithreading support. Esterel programs compiled for STARPro are significantly faster than Esterel software compilers, while achieving smaller code size at the same time. In comparison to an existing Esterel-optimized processor, KEP3a, STARPro achieves superior execution times, while suffering minimal code size penalties. This has been accomplished with a simpler hardware design, which at the same time, consumes significantly less hardware resources. The ability of the pipelined STARPro processor to operate at 167MHz in contrast to the non-pipelined operating frequency of KEP3a of only 60MHz further adds to the elegance of the STARPro approach.

Both the STARPro hardware and compiler have received minimal optimization at this stage. The hardware has a lot of room for further reduction in resource usage. The hardware scheduler, in particular, can be improved

to scale gate usage more optimally with increase in the number of supported threads.

On the compiler side, host procedures in Esterel (usually implemented in C) cannot yet be compiled directly to STARPro. It is envisioned that the STARPro compiler would eventually be able to support C data computation, as well as the reactive part of Esterel.

8 Acknowledgement

The authors would like to acknowledge the assistance of Rohan Aggrawal in STARPro's implementation, and Chia-Hao Chou for his contribution to the design.

References

[1] "Altera Corp." http://www.altera.com/products/ip/processors/nios2/ni2-index.html (Last Accessed: 20/11/2007).

[2] Berry, G., "The Constructive Semantics of Pure Esterel (Draft Book)," Available on-line, http://www-sop.inria.fr/esterel.org (Last Accessed: 28/4/2007), 1999.

[3] Berry, G., "The Esterel v5 Language Primer, Version v5_91," Centre de Mathématiques Appliquées, Ecole des Mines, Sophia-Antipolis (2000).

[4] Berry, G., "The Esterel v7 Reference Manual: Version v7_30 for Esterel Studio 5.3," Esterel Technologies SA, Villeneuve-Loubet, France (2005).

[5] Closse, E., M. Poize, J. Pulou, P. Venier and D. Weil, *Saxo-rt: Interpreting esterel semantic on a sequential execution structure*, Electronic Notes in Theoretical Computer Science **65** (2002), pp. 80–94.

[6] Dayaratne, M. W. S., P. S. Roop and Z. Salcic, *Direct execution of Esterel using reactive microprocessors*, in: *In Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, 2005.

[7] Edwards, S. A., "EstBench Esterel Benchmark Suit," http://www1.cs.columbia.edu/~sedwards/software.html (Last Accessed: 8/6/2007).

[8] Edwards, S. A. and J. Zeng, *Code generation in the Columbia Esterel Compiler*, EURASIP Journal on Embedded Systems **2007** (2007), pp. Article ID 52651, 31 pages, doi:10.1155/2007/52651.

[9] Harel, D. and A. Pnueli, *On the development of reactive systems*, in: K. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol. F-13 (1985), pp. 477–498.

[10] Li, X., M. Boldt and R. von Hanxleden, *Mapping Esterel onto a multi-threaded embedded processor*, in: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, 2006.

[11] Li, X., J. Lukoschus, M. Boldt, M. Harder and R. von Hanxleden, *An Esterel processor with full preemption support and its worst case reaction time analysis*, in: *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems* (2005), pp. 225–236.

[12] Li, X. and R. von Hanxleden, *The Kiel Esterel Processor - a semi-custom, configurable reactive processor*, in: S. A. Edwards, N. Halbwachs, R. v. Hanxleden and T. Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings (2005).

[13] Li, X. and R. von Hanxleden, *A concurrent reactive Esterel processor based on multi-threading*, in: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (2006), pp. 912–917.

[14] Plummer, B., M. Khajanchi and S. A. Edwards, *An esterel virtual machine for embedded systems*, in: *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, 2006.

[15] Potop-Butucaru, D. and R. de Simone, *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*, 2003, pp. 227–236.

[16] Salcic, Z., D. Hui, P. S. Roop and M. Biglari-Abhari, *ReMIC: design of a reactive embedded microprocessor core*, in: *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation* (2005), pp. 977–981.

[17] Schneider, K. and M. Wenz, *A new method for compiling schizophrenic synchronous programs*, in: *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems* (2001), pp. 49–58.

[18] Yoong, L. H., P. Roop, Z. Salcic and F. Gruian, *Compiling Esterel for distributed execution*, in: *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, 2006.