Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 610891, 19 pages
doi:10.1155/2009/610891

Research Article

A New Multithreaded Architecture Supporting Direct Execution of Esterel

Simon Yuan, Li Hsien Yoong, Sidharta Andalam, Partha S. Roop, and Zoran Salcic

Department of Electrical and Computer Engineering, University of Auckland, Auckland 1010, New Zealand

Correspondence should be addressed to Simon Yuan, iyua002@aucklanduni.ac.nz

Received 1 April 2008; Accepted 2 April 2009

Recommended by Marc Pouzet

We propose 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 preemption constructs, instructions for signal manipulation, and a notion of logical ticks for synchronous execution. In addition to the reactive processors, we propose a new intermediate format called unrolled concurrent control-flow graph with surface and depth ($UCCFG_{sd}$) that closely resembles the Esterel source. A compiler, based on $UCCFG_{sd}$, has been developed for code generation. We have synthesized STARPro and have carried out a range of benchmarking experiments. Experimental results reveal substantial improvement in performance and code size compared to software compilers. We also excel in comparison to recent reactive architectures, by achieving an average speed-up of 37% in worst-case reaction times and a speed-up of 38% in average-case reaction times. This has been achieved by utilizing fewer hardware resources, while incurring an average code size increase of 40%.

1. Introduction

The programming language Esterel [1] belongs to the family of synchronous languages [2]. Due to its synchronous semantics, all correct Esterel programs are guaranteed to be *reactive* and *deterministic* [3]. These properties greatly simplify the formal verification of programs, while at the same time, provide predictable run-time behavior. 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* [4].

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 nontrivial. We illustrate some aspects of this complexity using the example shown in Figure 1.

An Esterel program always consists of basic entities called modules. The example in Figure 1 has only a single module named `schizoCyc`. Within this module, an abort construct encloses the entire program, where input signal `R` can preempt the abort body at any instant, 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`.

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 [3]. 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 [5]. This

```
(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 imediate I;
(15)                emit A;
(16)                present C then emit D end;
(17)           end signal
(18)      end loop
(19) when R
(20) end module
```

Figure 1: The schizocyc example Esterel program.

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 Figure 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* behavior 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, nontrivial.

Several approaches exist for dealing with these complexities in the compilation and execution of Esterel programs. These include hardware compilation [3], software compilation for general-purpose microprocessors [6–8], and architecture-specific compilation for reactive processors optimized for Esterel [9, 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 additional 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 for Esterel execution, in contrast, 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 paper. We present a novel multithreaded processor, named Simultaneous multiThreaded Auckland Reactive Processor (STARPro), and an Esterel compiler for it, that achieves significant speed-up and code size compaction over traditional methods for software implementations of Esterel.

The rest of this paper is organized as follows. Section 2 reviews previous work related to architecture-specific execution of Esterel. Section 3 then presents STARPro's architecture, which is followed by a description of its instruction set architecture (ISA) in Section 4. Section 5 covers 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.

A preliminary version of the ideas in this manuscript has been presented in the 2008 workshop on Model-driven High-level Programming of Embedded Systems (SLA++P).

2. Related Work

The EMPEROR multiprocessor architecture [9] 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 [11]. 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 [10, 12–14]. 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 [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. At present, KEP3a employs a nonpipelined 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, this paper presents a novel multithreaded processor, named STARPro [15], 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 slightly inferior to KEP3a.

Plummer et al. [16] have explored another approach of executing Esterel using a virtual machine (VM). The VM provides customized instructions to support Esterel's execution, similar to STARPro. The key difference is that a virtual machine is implemented as software, whereas STARPro is a hardware platform. The code sizes in both approaches are superior when compared with traditional Esterel compilers. However, the VM approach is significantly slower than traditional Esterel compilers [16].

3. Architecture of the STARPro Processor

STARPro's design extends our previous reactive architecture REMIC [17]. 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, we have developed the Esterel Support Unit (ESU) to replace the RFU within REMIC, as illustrated in Figure 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 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 simultaneous multithreading (SMT) processor [18] 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, we will first explain how the datapath from REMIC is modified to work with TCB and AHB, before discussing how the two interact.

3.1. The Datapath. STARPro is an RISC reactive processor, featuring a three-state 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 [17] 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. In the following, we will explain in more detail how the ESU uses these new connections to the datapath.

3.2. 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 Figure 2(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 associated with the current thread. (The abort context will be described in Section 3.3.) 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 becomes 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 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 as boolean flags. We say that the *local tick* for a thread has elapsed whenever a `pause` statement 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 either have 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 has elapsed, 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 [9] 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 k *global tick* duration is

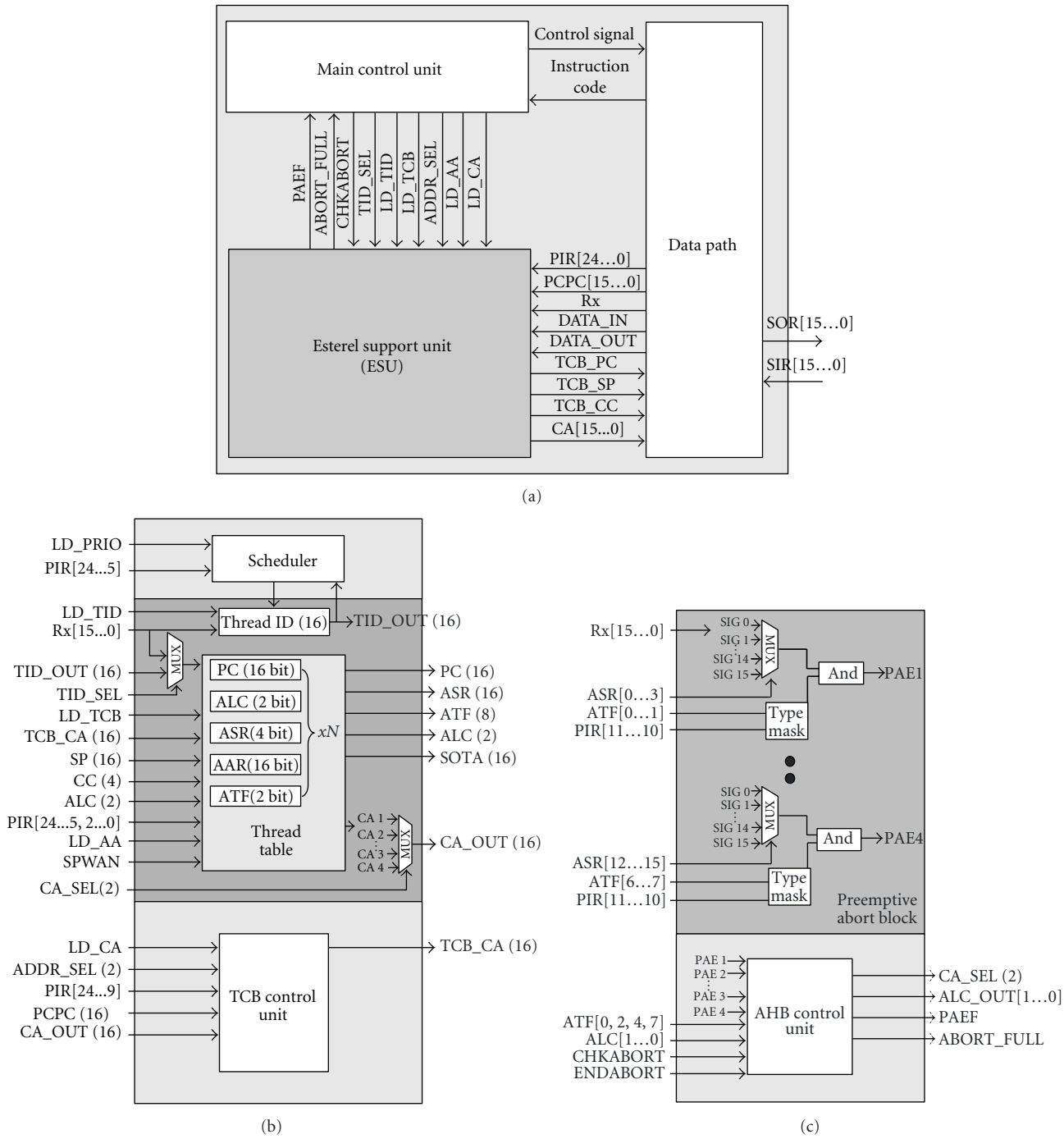


Figure 2: The STARPro architecture: (a) overview of hardware blocks, (b) thread Handling Block (TCB), and (c) abort Handling Block.

dynamically changed and is equal to the actual computational time required for executing a number of threads in any instant.

3.3. 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 over those nested below it. The AHB supports this feature by providing hardware-based priority resolution (controlled by a finite state machine in the AHB) for the abort constructs. The depth of nested aborts is fully parameterizable in our design. Figure 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.

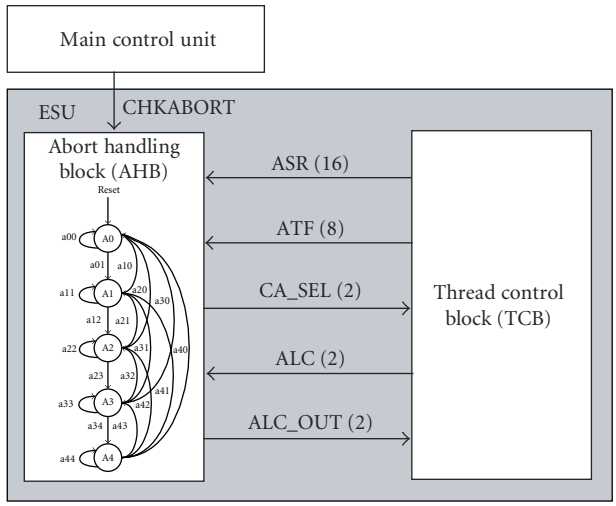


Figure 3: The interfacing between the AHB and the TCB.

(i) 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.

(ii) ASR (Abort Signal Register). This stores the ID of the signal which needs to be monitored during execution of an abort body.

(iii) AAR (Abort Address Register). This stores the continuation address, to which the thread must jump to should preemption happens.

(iv) 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 *nonimmediate*. These are orthogonal to each other, resulting in four distinct behaviors for abortions in Esterel.

(v) 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 this abort context of the current running thread to the AHB. The AHB does not contain any memory element, and it is purely control. When instructed by the main control unit, the AHB checks all abort levels that have been initialized. If a preemption is taken, it provides an index (CA_SEL in Figure 3) that selects the continuation address (AAR, stored in the thread table inside 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 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 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.

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

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

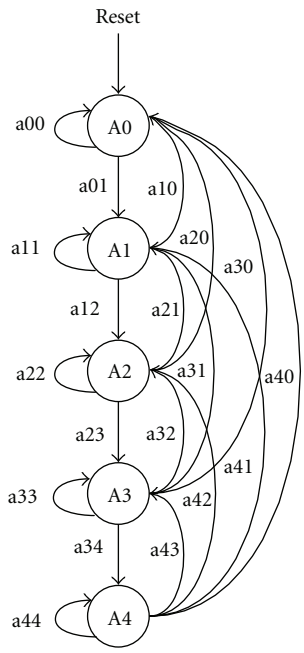
The distinction in behaviors of strong and weak abortions is controlled by the finite state machine inside the control unit of the AHB. Based on the type of abortion, the condition of a transition triggered between states changes. The FSM depicted in Figure 4(a) contains five states for four levels of aborts, and the transition condition is summarized in Figure 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 abortion specified by CHKABORT instruction is stored as a single bit in the instruction operand. The operand is passed through the Pipelined Instruction Register (PIR) 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 Preemptive Abort Event (PAEs) are present at the same time, PAE1 would be taken, as it is given higher priority over others by the transition condition for strong aborts (see the left hand side of Figure 4(b)). Again, at state A4, if the abort type given by the CHKABORT instruction is weak, a transition to A0, for example, can only happen if PAE1 is the only PAE signal present (see transition a40 on the right hand side of Figure 4(b)). If PAE1 and PAE4 are both present, then a transition via a43 would be taken.

We describe the reason for this difference. When weak aborts are nested, the bodies of the weak abort that took the preemption will be given one last chance to complete the current *tick*. To preserve such semantics, we designed the AHB to preempt weak aborts from inside out. This is in contrast to nested strong aborts, where the abort bodies of a strong abort are preempted at the beginning of the *tick*. In order for the AHB to start the preemption of weak aborts inside out, the state machine of the AHB differentiates weak aborts from the strong.

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



(a)

State Description:

A0: Abort empty (no ABORT loaded)
A1: Abort Level 1 (AASR1 and AAAR1 loaded with 1st ABORT)
A2: Abort Level 2 (AASR2 and AAAR2 loaded with 2nd ABORT)
A3: Abort Level 3 (AASR3 and AAAR3 loaded with 3rd ABORT)
A4: Abort Level 4 (AASR4 and AAAR4 loaded with 4th ABORT)

State Transition Condition (strong):	**State Transition Condition (weak):**
a00: LD_AA = '0'	a00: LD_AA = '0'
a01: LD_AA = '1'	a01: LD_AA = '1'
a10: (ENDABORT = '1' or PAE1 = '1')	a10: (ENDABORT = '1' or PAE1 = '1')
a11: No operation	a11: No operation
a12: LD_AA = '1'	a12: LD_AA = '1'
a20: PAE1 = '1'	a20: PAE1 = '1' and PAE2 = '0'
a21: (ENDABORT = '1' or PAE2 = '1')	a21: (ENDABORT = '1' or PAE2 = '1')
a22: No operation	a22: No operation
a23: LD_AA = '1'	a23: LD_AA = '1'
a30: PAE1 = '1'	a30: PAE1 = '1' and PAE2 = PAE3 = '0'
a31: PAE2 = '1'	a31: PAE2 = '1' and PAE3 = '0'
a32: (ENDABORT = '1' or PAE3 = '1')	a32: (ENDABORT = '1' or PAE3 = '1')
a33: No operation	a33: No operation
a34: LD_AA = '1'	a34: LD_AA = '1'
a40: PAE1 = '1'	a40: PAE1 = '1' and PAE2 = PAE3 = PAE4 = '0'
a41: PAE2 = '1'	a41: PAE2 = '1' and PAE3 = PAE4 = '0'
a42: PAE3 = '1'	a42: PAE3 = '1' and PAE4 = '0'
a43: (ENDABORT = '1' or PAE4 = '1')	a43: (ENDABORT = '1' or PAE4 = '1')
a44: No operation	a44: No operation

(b)

Figure 4: Finite State Machine of the Abort Handling Block: (a) the FSM, and (b) state transition conditions.

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. Later, in Section 5.2, we will discuss in more detail how the four types of abortions in Esterel are handled for execution on STARPro.

In comparison to the design of preemption mechanism in KEP, the most significant difference between the AHB and the preemption *watchers* in KEP [10] is how the preemption is monitored. STARPro relies on explicit checks at appropriate times using an instruction, whereas the *watchers* in KEP 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 behavior. The difference between the two approaches results in simpler preemption hardware design for the STARPro. However, the *watcher* design of KEP 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 the limitation, our compiler is able to handle aborts in software in addition to hardware 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 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.

STARPro also does not have any dedicated instruction for strong immediate aborts. Instead, this is implemented 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 Figure 1. The equivalent STARPro assembly code for that example is shown in Figure 5. For the reader's convenience, Figure 1 is replicated as Figure 6 next to Figure 5. We start 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 3, the first abort level is configured here to watch for signal 14 (signal `R`). Then, the program forks two concurrent threads. This is accomplished

```
 (1)      LDR R6 $INPUTS ; external inputs
 (2)      ; -------------- Start of program --------------
 (3)      ABORT S14 L13 ; abort S14=R
 (4) L0   LDR R0 $SIGNALS
 (5)      CBIT R0 R0 #A
 (6)      STR R0 $SIGNALS
 (7)      LDR R0 #1   ; create
 (8)      SPAWN R0 T1 ; thread 1
 (9)      PCHANGE R0 #0 ; assign thread 1 with priority 0
(10)      LDR R0 #2   ; create
(11)      SPAWN R0 T2 ; thread 2
(12)      PCHANGE R0 #0 ; assign thread 2 with priority 0
(13)      LDR R0 #31   ; special thread for
(14)      SPAWN R0 GTK ; handling global ticks
(15)      LDR R0 #$6   ; set threads to
(16)      STR R0 $JOIN ; NOT join
(17)      CSWITCH #255
(18)      ; after join
(19)      CHKABORT R6 STRONG
(20)      JMP L0
(21) L13  JMP EN
(22)      ; -------- Start of global tick handler --------
(23) GTK  LDR R7 #0       ; clear
(24)      STR R7 $OUTPUTS ; outputs
(25)      LDR R6 $INPUTS ; new snapshot of inputs
(26)      CSWITCH #255
(27)      JMP GTK
(28)      ; ------------- Start of thread 1 --------------
(29) T1   ABORT S14 L6 ; abort S14=R
(30)      CSWITCH #2
(31)      LDR R0 $SIGNALS
(32)      PRESENT $15 R0 L5 ; present S15=A
(33)      SBIT R7 R7 #B ; emit B
(34)      STR R7 $OUTPUTS
(35) L5   PAUSE #0
(36)      CHKABORT R6 STRONG
(37)      CSWITCH #1
(38)      PRESENT S15 R7 L4 ; present S15=B
(39)      LDR R0 $SIGNALS
(40)      SBIT R0 R0 #A ; emit A
(41)      STR R0 $SIGNALS
(42) L4   PAUSE #0
(43)      CHKABORT R6 STRONG
(44)      CSWITCH #3
(45)      LDR R0 $SIGNALS
(46)      PRESENT S15 R0 L4 ; present S15=A
(47) L3   SBIT R7 R7 #C ; emit C
(48)      STR R7 $OUTPUTS
(49)      ENDABORT
(50) L6   LDR R0 $JOIN  ; mark
(51)      CBIT R0 R0 #1 ; thread 1
(52)      STR R0 $JOIN  ; dead
(53)      SZ L1 ; threads join if JOIN == 0
(54)      JMP L2
(55) L1   LDR R0 #0 ; activate the parent thread
(56)      PCHANGE R0 #5
(57) L2   CSWITCH #255 ; set current thread to inactive
(58)      ; ------------- Start of thread 2 --------------
(59) T2   ABORT S14 L12 ; abort S14=R
(60)      ABSENT S15 L10 ; absent S15=I
(61) L11  PAUSE #0
(62)      CHKABORT R6 STRONG
(63)      PRESENT S15 R6 L11 ; present S15=I
(64) L10  LDR R0 $SIGNALS
(65)      SBIT R0 R0 #A ; emit A
(66)      STR R0 $SIGNALS
(67)      CSWITCH #4
(68)      PRESENT S14 R7 L9 ; present S14=C
(69)      SBIT R7 R7 #D ; emit D
(70)      STR R7 $OUTPUTS
(71) L9   ENDABORT
(72) L12  LDR R0 $JOIN  ; mark
(73)      CBIT R0 R0 #2 ; thread 2
(74)      STR R0 $JOIN  ; dead
(75)      SZ L7 ; threads join if JOIN == 0
(76)      JMP L8
(77) L7   LDR R0 #0 ; activate the parent thread
(78)      PCHANGE R0 #5
(79) L8   CSWITCH #255 ; set current thread to inactive
(80) EN   END
```

Figure 5: The `schizoCyc` example translated to STARPro assembly.

Table 1: Esterel-Oriented instructions.

Instruction Syntax	Description
`SPAWN` *Reg StartAddr*	Creates a new thread
`CSWITSH` *PriorityVal*	Updates the priority of the currentthread (which executes the CWITCH) to the value of *PriorityVal*. It then passes control to the scheduler that has to select the next highest priority thread for execution
`PAUSE` *PriorityVal*	Same as `CSWITCH`, in addition it marks the end of local tick
`PCHANGE` *Reg PriorityVal*	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

```
 (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 imediate I;
(15)                 emit A;
(16)                 present C then emit D end;
(17)            end signal
(18)       end loop
(19) when R
(20) end module
```

Figure 6: The `schizoCyc` example Esterel program.

using the `SPAWN` instruction on lines 8 and 11, which creates two new entries in the thread table of the TCB for threads 1 and 2. These threads are then initialized to start at labels `T1` and `T2`, respectively. Line 14 creates the special *global tick* management thread. The `PCHANGE` instruction on lines 9 and 12 sets the initial priority of threads 1 and 2. Finally, the `CSWITCH` instruction on line 17 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. 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 next instruction cycle. Since both threads 1 and 2 have the same priority, the scheduler can randomly select either thread for execution first. The PAUSE instruction functions similarly to CSWITCH, but in addition, also marks the end of a *local tick* for the thread that executed it. PAUSE instructions can be found on several lines across threads 1 and 2.

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 [9]. Due to this, thread 1 and thread 2 begin with an ABORT instruction on lines 28 and 58, 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 35 and 42. 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 48 and 71) 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 32, 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. Our compiler inserts the appropriate conditional branch by anticipating the most probable branch of the condition. For example, when the number of nested aborts exceeds what the hardware had been synthesized with, the compiler inserts ABSENT instructions in place of CHKABORT. If a 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 PRESENT instruction is taken, the processor pipeline will be flushed, which reduces the performance benefit of pipelining.

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.

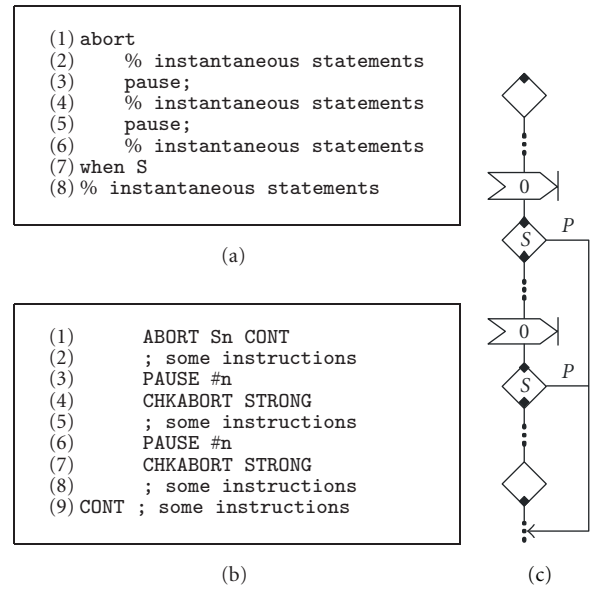


FIGURE 7: Mapping of a strong abort: (a) esterel source, (b) assembly, and (c) UCCFG$_{sd}$.

5.1. Unrolled Concurrent Control-Flow Graph. The UCCFG$_{sd}$ is a variant of the UCCFG intermediate format, which was first introduced in [9]. However, the UCCFG is not capable of fully preserving Esterel's semantics, especially for statements that have distinct start and resumption behaviors (also known as *surface* and *depth* behaviors), like that of the abort statement described in the example of Figure 1. Some statements, like emit, are logically instantaneous, while others, like the await statement, consume time (*ticks*). Such noninstantaneous statements have distinct surface and depth behaviors.

To overcome this, we have modified the original UCCFG format and extended it to explicitly capture both the surface and depth behavior of every statement in Esterel. This approach adapts the technique used in [7], where the start and resumption behaviors are differentiated using distinct *surface code* and *depth code*.

However, unlike [7], the control flow graph in our case is unrolled due to explicit representation of the *ticks*. Moreover, we have no explicit state encoding using state variables. Hence, our intermediate format is simpler and closely resembles the Esterel source.

In [7], 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 Figure 11. Using these *pause* nodes, the loop required to

execute the CFG of [7] can be completely unrolled. Hence, instead of using a switch statement to select between the surface and depth code as done in [7], code for STARPro can be conveniently represented in the following form:

Surface (`code`) `followed by` *depth* (`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.

5.2. Translating Aborts. Translation of aborts is 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.

(i) 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.

(ii) 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.

(iii) 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.

(iv) The *nonimmediate* 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 described below.

In the following, we further elaborate how the four types of aborts are translated. Depicted in Figure 7 is an example of strong abort. The abort body consists of a series of sequential statements, with a *pause* statement in between each pair of instantaneous statements. These instantaneous statements are denoted as "· · · " in Figure 7(c).

The `abort` and `when S` statements are directly mapped to the *abort start* and *abort end* nodes. These two nodes are drawn as diamonds with a single dot at the top and bottom corners, respectively. In between these two nodes is the abort body. Nodes following the *abort end* nodes correspond to the statements of the abort handler, if one exists. Otherwise, these are the continuation context after the abort. Within the abort body, each *pause* statement is mapped to a *pause* node. After each *pause* node, a *check abort* node is inserted immediately below. By mapping each node in the UCCFG_{sd} to assembly, we obtain the final result in Figure 7(b).

The assembly version very closely resembles 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.

The immediate version of a strong abort, in addition, checks for preemption at the beginning of the starting instant (surface behavior). This requires only a *signal test* node before entering the abort body (see Figure 8(c)).

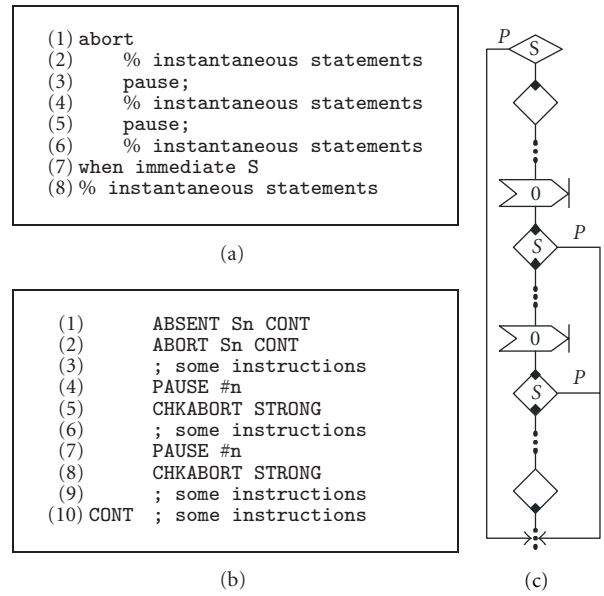


Figure 8: Mapping of a strong immediate abort: (a) esterel source, (b) assembly, and (c) UCCFG_{sd}.

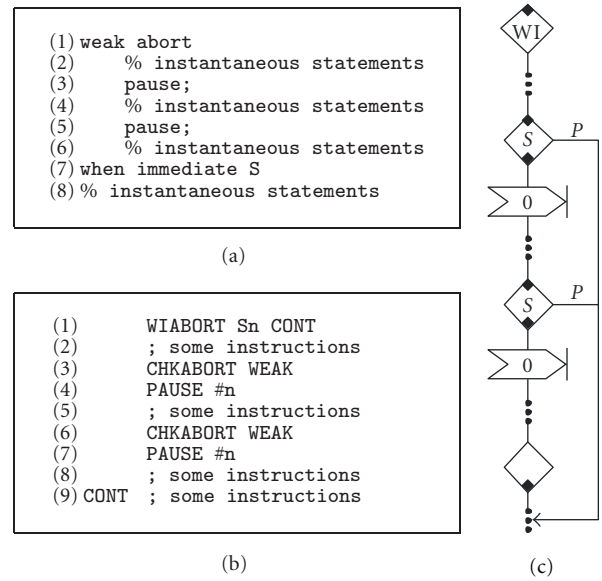


Figure 9: Mapping of a weak immediate abort: (a) esterel source, (b) assembly, and (c) UCCFG_{sd}.

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 behavior, checking for preemption is done at the end of each tick. Figure 9(c) illustrates how a *check abort* node is inserted immediately above each *pause* node.

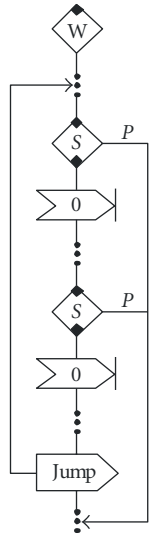
The handling of a nonimmediate weak abort is subtle when the abort body contains a loop. The first pass through the loop is different from all subsequent passes, as the *surface* part of the loop body gets folded back into the *depth* after the first pass. In this case, the abortion condition needs not

```
(1) weak abort
(2)   loop
(3)     % instantaneous statements
(4)     pause;
(5)     % instantaneous statements
(6)     pause;
(7)     % instantaneous statements
(8)   end loop
(9) when S
(10) % instantaneous statements
```

(a)

```
(1)       WABORT Sn CONT
(2) LOOP  ; some instructions
(3)       CHKABORT WEAK
(4)       PAUSE #n
(5)       ; some instructions
(6)       CHKABORT WEAK
(7)       PAUSE #n
(8)       ; some instructions
(9)       JMP LOOP
(10) CONT ; some instructions
```

(b)



(c)

Figure 10: Mapping of a weak abort: (a) esterel source, (b) assembly, and (c) $UCCFG_{sd}$.

to 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 nonimmediate aborts using an additional status bit, which we refer to as the *surface flag*. The *surface flag* is only valid for nonimmediate weak aborts, and it is initialized to false by a `WABORT` instruction, indicating that the surface of the body has not been executed. The *surface flag* is set on the first `CHKABORT` instruction in the nonimmediate weak abort body, and the hardware skips over this first `CHKABORT`. The `CHKABORT` instruction will only work after the *surface flag* is set for nonimmediate weak aborts. This explains why Figures 10(b) and 10(c) are exactly the same as their immediate counterpart. The *abort start* node with the letter **W** inside is the only clue that the abort is a nonimmediate version.

Note the orthogonality of the four types of aborts. For example, the translation for strong-immediate (**SI**) abort is different from that of the weak-immediate (**WI**). Unlike the **SI** that has an absent statement guarding the abort to deal with preemption at the starting instant, we do not have a similar strategy for **WI**. This is because in case of **WI** preemption, if the preemption is taken, this can happen only after executing instantaneous code inside the body before the first *pause* statement.

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 local signal may potentially assume multiple statuses within the *tick*. Such programs are referred to as *schizophrenic* [3, 5]. This phenomenon may result in a single local signal declaration

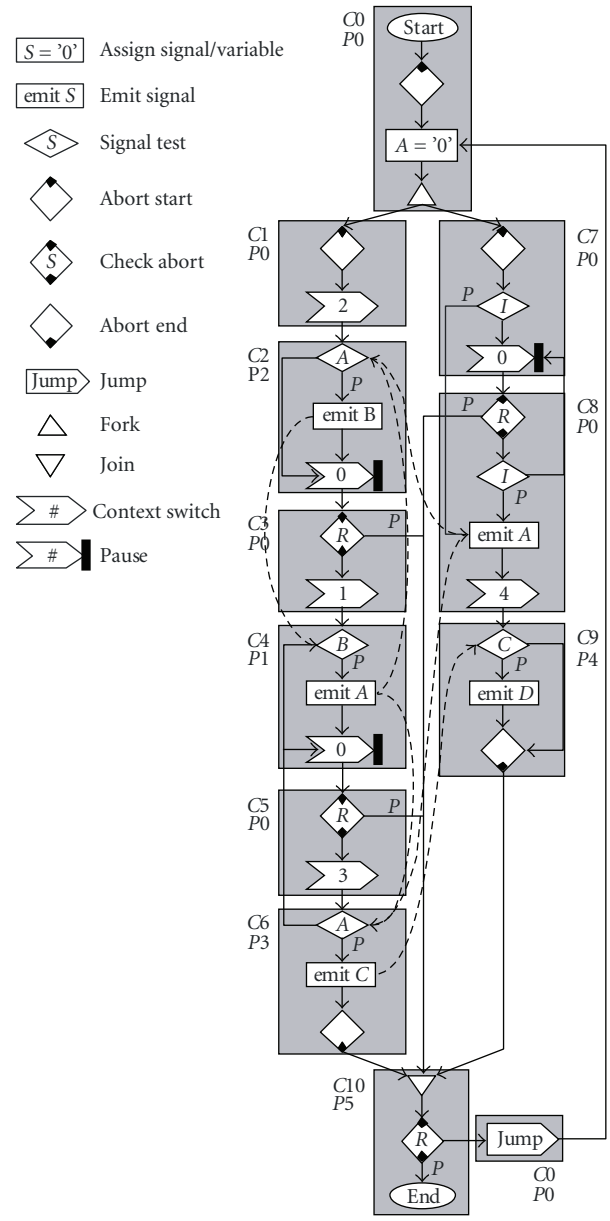


Figure 11: Unrolled Concurrent Control Flow Graph of the `SchizoCyc` example.

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* [3]) 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. 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 redeclared. 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 Figure 11. The *join* node acts as a rendezvous point for the two threads, ensuring that they will always join before jumping back to the top of the loop to create a new copy of signal `A`.

5.4. Code Generation. The `SchizoCyc` example contains a strong abortion. In Figure 11, 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.

In Esterel, when a thread gets preempted, its sibling threads are also synchronously preempted in the same instant. In order to preserve the same behavior for STARPro, a *check abort* node is inserted immediately below the *join* node. By doing so, when the preempting signal becomes present, each of the two threads in the example takes the present branch (the preempting signal is present) of the *check abort* node inside the forked threads to the *join* node. Following the conjunction of the two threads at the *join* node, the *check abort* node below the *join* node allows the parent thread to also check for preemption and reacts to the preemption synchronously in the same instant with all its sibling threads.

The nodes in the UCCFG$_{sd}$ map very closely to STARPro instructions. Backend 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 Figure 11 are the *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 7 to 17 in Figure 5 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 `SchizoCyc` example, one thread could finish before the other. When a thread first 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 50 ∼ 57 and 72 ∼ 79 in Figure 5. When all threads are ready to join (the JOIN variable evaluates to zero), whichever is, 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 in hardware will select the parent thread. In the next subsection, we will explain how scheduling is done.

5.5. Scheduling. Scheduling of threads is done in hardware at run-time based on the priorities of the threads precomputed at compile-time. We will first explain how the UCCFG$_{sd}$ is traversed before presenting the scheduling algorithms.

The scheduling algorithms traverse the UCCFG$_{sd}$ by following two kinds of paths, either a control arc or a data dependency arc. For any given node in the UCCFG$_{sd}$ connected to other nodes by a control arc, we refer to nodes immediately preceding the current one as *control predecessors* and nodes immediately succeeding the current one as *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 Figure 11 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 *signal test* node on the output signal `C` found in the thread on the right branch. Its data predecessor would be the *emit* node to output signal `C` found in the thread on the left branch.

In the next four subsections, we will present three algorithms that are used to determine how the threads are to be interleaved, and how a schedule is followed at run-time. 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 *cswitch* nodes: this algorithm inserts *cswitch* nodes when necessary based on the priority assigned to the clusters.

(4) Scheduling in hardware: this subsection explains how threads are interleaved at run-time, and how the hardware orders the threads.

5.6. Clustering. The first step to schedule threads 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 was inspired by CEC's [8], and has been modified so as to adapt to our intermediate format. We present the clustering algorithm in Figure 12.

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 move a

```
(1) C denotes a set of clustered nodes
(2) C_s denotes the cluster of the control successor of a node
(3) C_i denotes the current cluster being worked on
(4) i = 0
(5) add topmost CF node to F, the frontier set
(6) while F≠∅ do
(7)       arbitrarily select and remove f from F
(8)       create a new, empty pending set P
(9)       add f to P
(10)      set C_i to the empty cluster
(11)      while P≠∅ do
(12)            randomly select and remove p from P
(13)            if p ∉ C and (p has no data predecessors or
                C_i = ∅) then
(14)                  add p to C_i and C
(15)                  if p = jump then
(16)                        if p's successor ∈ C_s then
(17)                              insert p into C_s and C
(18)                        else
(19)                              insert p into F
(20)                              remove p from C_i and C
(21)                              add all of p's control successors to P
(22)                        end
(23)                  else if p≠ fork and p≠ threadswitch then
(24)                        add all of p's control successors to P
(25)                  end
(26)                  add all of p's control successors to F
(27)            end
(28)            if p ∈ C then
(29)                  remove p from F
(30)            end
(31)      end
(32)      if C_i≠∅ then
(33)            i = i + 1
(34)      end
(35) end
```

Figure 12: The clustering algorithm.

node from F into P. The outer loop creates a new cluster C_i everytime P is emptied by the inner loop, and 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 predecessor. 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 Figure 11, 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 is emptied, the outer loop restarts, creating a new cluster C_1 and adding one of the *abort start* nodes to it.

Lines 15 $\sim$ 22 in the algorithm specially looks for *jump* nodes, where a *jump* node is a node that unconditionally jumps to some location in the program. It is used, for example, to implement a loop. When a *jump* node leads to a node that has a data predecessor, a *cswitch* node has to be inserted before the *jump* node. This means that 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 adding its successor to P. This way, a *jump* node is always clustered after its control successor has been clustered. If a node p is not a *jump* and it is neither a *fork* nor a *threadswitch* (where a *threadswitch* node is either a *pause* or *cswitch*), then all of p's control successors are added to set P.

```
(1) module InstCyc:
(2) inpute I;
(3) outpute A, B;
(4) present I then
(5)       present A then emit B end;
(6) end
(7) ||
(8) present I else
(5)       present B then emit A end;
(10) end
(11) end module
```

Figure 13: An example of a causal program with a false instantaneous cyclic dependency.

Table 2: Hardware Resource Usage–STARPro versus KEP3a.

STARPro @167 MHz	Max. Threads	2	4	8	16	32	64	128	256	512
	Gates (K)	24	24	26	29	52	89	173	342	682
KEP3a @60 MHz	Max. Threads	2	10	20	40	60	80	100	120	
	Gates (K)	295	299	311	328	346	373	389	406	

5.7. 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 consumers. 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 noncausal programs. We can, however, generate correct code for programs with certain types of cyclic dependencies.

In Esterel, only noninstantaneous cyclic dependencies are allowed [3]. `SchizoCyc` is an example of this. It is also possible that an instantaneous cyclic dependency seemingly exists in a program, but the dependency, in fact, does not exist because at least one of the nodes in the cycle can never be reached in the same instant as the rest. An example of such a program is shown in Figure 13. 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, our compiler can still generate correct code. In such a case, the priority assignment assumes that the program is causal, and an arbitrary cluster will be chosen as a starting point.

Table 3: A list of example Esterel programs used.

Module name	Lines of code	No. of threads
abcd	101	5
abcdef	119	7
eight_but	137	9
chan_prot	55	6
reactor_ctrl	32	4
runner	53	3
example	21	3

```
(1) foreach cluster C in the program do
(2)     traceDataPred(C)
(3) end

(1) function traceDataPred(C)
(2)     if C is visited then return priority of C
(3)     add C to the visited set
(4)     max_depth = 0
(5)     foreach CF node n in C do
(6)         depth = 0
(7)         foreach data predecessor p of n do
(8)             if n = join then
(9)                 n' = first non-jump control predecessor of p
(10)                depth = traceDataPred(cluster of n') +1
(11)                if depth > max_depth then
(12)                    max_depth = depth
(13)                end
(14)            else if p ∉ C then
(15)                depth = traceDataPred(cluster of n) +1
(16)                if depth > max_depth then
(17)                    max_depth = depth
(18)                end
(19)            end
(20)        end
(21)    end
(22)    assign priority of C with max_depth
(23)    return max_depth
(24) end
```

Figure 14: The priority assignment algorithm.

Our compiler relies on existing tools [19] to do a priori causality analysis prior to compilation. This step is needed to ensure correct code generation using our compiler.

We present the priority assignment algorithm in Figure 14. 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 2 prevents the algorithm from deadlocking in a dependency cycle. For an unvisited cluster, it is, by default, assigned with a priority of 0 if no data predecessors can be found. The default priority comes from the *max_depth* variable on line 4 in `traceDataPred`.

The loop on line 5 traverses through every control flow nodes 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 `traceDataPred`.

A *join* node is specially handled on lines 8 ∼ 13. This is required because control cannot flow immediately to the join node. Instead, all joining threads need to terminate

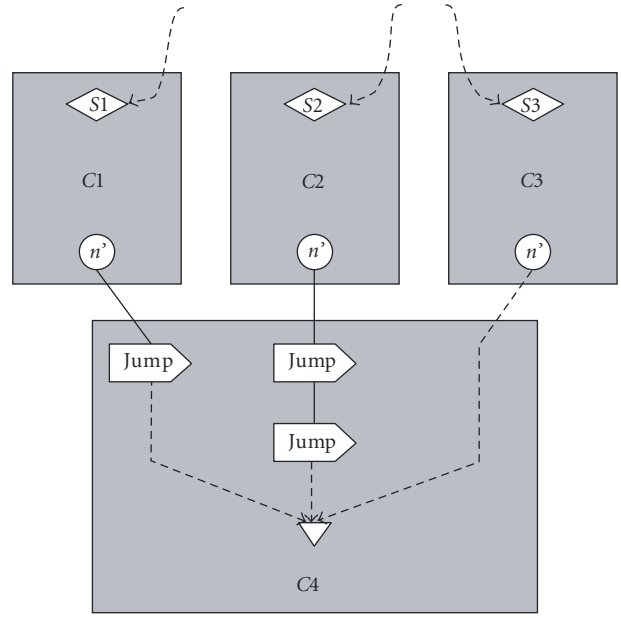


Figure 15: Tracing upwards from a join node.

synchronously before the join node can be reached. To enforce such execution order of the *join* node, the control arcs from the control predecessors of a *join* node are replaced with data dependency arcs. The priority assignment algorithm can then be used to assign the cluster of the *join* node a lower priority than its preceding clusters.

As mentioned earlier, a *jump* node is grouped into the cluster of its control successor. In the case of a *join* node, it is possible that a *jump* node or a sequence of consecutive *jump* nodes precedes the *join* node and resides in the same cluster. Moreover, because the control arcs from these *jump* nodes have been replaced with data dependency arcs, and if an incoming data dependency arc of a *join* node comes from a *jump* node, tracing the arc will lead to a node in the same cluster. Figure 15 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 Figure 15 a priority, line 9 in the algorithm traverses the UCCFG_{sd} upward until it finds a non*jump* node n'; then on line 10, the cluster of n' is passed to `traceDataPred` to derive a priority from the preceding clusters of C4. These would be C1 and C2 in Figure 15.

The priority of each *chain* of data dependency arcs are incremented by one, and then the maximum priority value of the deepest data dependency chain is assigned to *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 10 ∼ 13 and lines 15 ∼ 18. 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 Figure 11. Assume that C0 and C1 have been visited, and C2 is the first cluster `traceDataPred` found to have dependencies. C2 has two incoming dependency arcs, one from C8 and another from C4. Assume that

```
(1) foreach cluster C in the program do
(2)     foreach CF node n in C do
(3)         if n = pause then
(4)             store the priority of the cluster n's successor
                belongs to in the pause node
(5)             continue
(6)         end
(7)         if n = fork then continue
(8)         foreach control successor s of n do
(9)             if priority of the cluster of s > C's priority then
(10)                insert a cswitch node between n and s
(11)                store the priority of the cluster s belongs
                    to in the cswitch node
(12)            end
(13)        end
(14)    end
(15) end
```

Figure 16: The *cswitch* insertion algorithm.

the algorithm recursively calls `traceDataPred` on C4 first. Cluster C4 has only one incoming dependency arc from C2, creating a cycle. The arc is traced by `traceDataPred` (*C*2). Since C2 is already visited and currently holds a priority of 0, `traceDataPred` (*C*2) immediately returns 0. The *depth* variable in `traceDataPred` (*C*4) is assigned with the priority of C2 plus 1; *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; so `traceDataPred` (*C*4) returns 1. At this point, C2 still has one more dependency arc from C8. This is again traced by calling `traceDataPred` (*C*8). 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*, and hence, C2 is now permanently assigned with a priority of 2 (*max_depth* + 1).

5.8. Inserting Cswitch Nodes. To interleave between threads, *cswitch* nodes have to be inserted between transitions from one cluster to another. The final algorithm presented in Figure 16 does this. This is done by the *cswitch* 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 ∼ 5 of the algorithm use 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 7 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 8 ∼ 9 ensure that a minimum number of *cswitch* nodes are inserted. A *cswitch* 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 *cswitch* node is not necessary, as the lower priority value of the succeeding cluster means that 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 11 stores the priority of the succeeding cluster in each *cswitch* node.

We will again use Figure 11 to illustrate how *cswitch* nodes are inserted by the algorithm. Starting with the first cluster C0, it has two exit points from *fork*. Exit points from a *fork* node are skipped by the algorithm. The next cluster C1 visited by the algorithm has one exit point to C2. The transition from C1 to C2 is a transition from a higher priority cluster to that of a lower one (from P0 to P2). A *cswitch* is inserted before C2 to give any potential signal emitters a chance to execute prior to C2. The newly inserted *cswitch* node has a priority value of 2 stored in it, where the value comes from the priority of C2. Moving on to the next cluster C2, the *pause* node in C2 is assigned with the priority of C3. One thing that Figure 11 does not show is that the control arcs leading to the *join* node have actually been replaced with data dependency arcs. Because of this change, the algorithm does not find exit points from clusters that previously had control arcs to *join*. Hence, no *cswitch* is inserted, and instead, context switching and reviving of the parent thread is handled by the *join* node.

5.9. Scheduling in Hardware. The priority of a thread is changed by executing the `PCHANGE`, `CSWITCH`, or `PAUSE` instruction. 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 *local tick* status flags of every thread. The scheduler makes a decision on which thread to select when a `CSWITCH` or `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 *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 global *tick* handler thread is selected, and the *local tick* flags of every thread are reset to false. This completes the global tick.

The last step described above is only performed at the end of a global *tick*, which can only be reached when the *local tick* status flags of all active threads evaluate to true. At the end of each 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 *global tick* handler. The scheduler selects this thread when step (4) is performed. However, as a single threaded program does not rely on a special global *tick* handler, the code of the *global tick* handler is simply generated for the *pause* nodes instead.

To illustrate how `SchizoCyc` executes on STARPro, we show the same assembly code in Figure 17(a) with comments

removed. On the right, Figure 17(b) shows the changes of the values of the program counter, *local tick* status flag, and the priority of each thread.

In the starting instant of the program, the root thread starts forking into two threads. The global *tick* handler thread gets created after line 12, while threads 1 and 2 are created on lines 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 reassigned by the `PCHANGE` instruction, as can be seen in the table at the top of Figure 17(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 the 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 16 until the thread is resumed. The same applies to all other threads that are suspended through context switches.

Continuing on, since both threads 1 and 2 have the same priority, it does not matter which one is executed first. 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 threads 1 and 2 have not completed their *local ticks* and are still active; thus both are valid candidates to be scheduled next. Since thread 2 now has higher priority, it is selected for execution.

The fourth table shows the result after executing thread 2 up to the first `PAUSE` instruction on line 55, assuming that 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 1 yet to complete its *local tick*. Similarly, we arrive with the results 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 threads 1 and 2 have both completed their *local ticks*. This triggers a global *tick* signal 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 22. Note that the *local tick* status flags have been reset to false. The flow of the execution carries on in similar fashion as described above.

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, 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

```
 (0)       LDR R6 $INPUTS
 (1)       ABORT S14 L13
 (2) L0    LDR R0 $SIGNALS
 (3)       CBIT R0 R0 #A
 (4)       STR R0 $SIGNALS
 (5)       LDR R0 #1
 (6)       SPAWN R0 T1
 (7)       PCHANGE R0 #0
 (8)       LDR R0 #2
 (9)       SPAWN R0 T2
(10)       PCHANGE R0 #0
(11)       LDR R0 #31
(12)       SPAWN R0 GTK
(13)       LDR R0 #$6
(14)       STR R0 $JOIN
(15)       CSWITCH #255
(16)       CHKABORT R6 STRONG
(17)       JMP L0
(18) L13   JMP EN
(19) GTK   LDR R7 #0
(20)       STR R7 $OUTPUTS
(21)       LDR R6 $INPUTS
(22)       CSWITCH #255
(23)       JMP GTK
(24) T1    ABORT S14 L6
(25)       CSWITCH #2
(26)       LDR R0 $SIGNALS
(27)       PRESENT S15 R0 L5
(28)       SBIT R7 R7 #B
(19)       STR R7 $OUTPUTS
(30) L5    PAUSE #0
(31)       CHKABORT R6 STRONG
(32)       CSWITCH #1
(33)       PRESENT S15 R7 L4
(34)       LDR R0 $SIGNALS
(35)       SBIT R0 R0 #A
(36)       STR R0 $SIGNALS
(37) L4    PAUSE #0
(38)       CHKABORT R6 STRONG
(39)       CSWITCH #3
(40)       LDR R0 $SIGNALS
(41)       PRESENT S15 R0 L4
(42) L3    SBIT R7 R7 #C
(43)       STR R7 $OUTPUTS
(44)       ENDABORT
(45) L6    LDR R0 $JOIN
(46)       CBIT R0 R0 #1
(47)       STR R0 $JOIN
(48)       SZ L1
(49)       JMP L2
(50) L1    LDR R0 #0
(51)       PCHANGE R0 #5
(52) L2    CSWITCH #255
(53) T2    ABORT S14 L12
(54)       ABSENT S15 L10
(55) L11   PAUSE #0
(56)       CHKABORT R6 STRONG
(57)       PRESENT S15 R6 L11
(58) L10   LDR R0 $SIGNALS
(59)       SBIT R0 R0 #A
(60)       STR R0 $SIGNALS
(61)       CSWITCH #4
(62)       PRESENT S14 R7 L9
(63)       SBIT R7 R7 #D
(64)       STR R7 $OUTPUTS
(65) L9    ENDABORT
(66) L12   LDR R0 $JOIN
(67)       CBIT R0 R0 #2
(68)       STR R0 $JOIN
(69)       SZ L7
(70)       JMP L8
(71) L7    LDR R0 #0
(72)       PCHANGE R0 #5
(73) L8    CSWITCH #255
(74) EN    END
```

(a)

After line 12 on tick 0

Thread	PC	Ticked	Priority
0	13	N	0
1	24	N	0
2	53	N	0
31	19	N	N/A

After line 15 on tick 0

Thread	PC	Ticked	Priority
0	16	N	255
1	24	N	0
2	53	N	0
31	19	N	N/A

After line 25 on tick 0

Thread	PC	Ticked	Priority
0	16	N	255
1	26	N	2
2	53	N	0
31	19	N	N/A

After line 55 on tick 0

Thread	PC	Ticked	Priority
0	16	N	255
1	26	N	2
2	56	Y	0
31	19	N	N/A

After line 30 on tick 0

Thread	PC	Ticked	Priority
0	16	N	255
1	31	Y	0
2	56	Y	0
31	19	N	N/A

After line 22 on tick 1 I

Thread	PC	Ticked	Priority
0	16	N	255
1	31	N	0
2	56	N	0
31	23	N	N/A

After line 32 on tick 1 I/A

Thread	PC	Ticked	Priority
0	16	N	255
1	33	N	1
2	56	N	0
31	23	N	N/A

After line 61 on tick 1 I/A

Thread	PC	Ticked	Priority
0	16	N	255
1	33	N	1
2	62	N	4
31	23	N	N/A

After line 37 on tick 1 I/A

Thread	PC	Ticked	Priority
0	16	N	255
1	38	Y	0
2	62	N	4
31	23	N	N/A

After line 73 on tick 1 I/A

Thread	PC	Ticked	Priority
0	16	N	255
1	38	Y	0
2	74	N	255
2	23	N	N/A

(b)

Figure 17: Change of thread context for SchizoCyc example: (a) assembly, and (b) change of thread context.

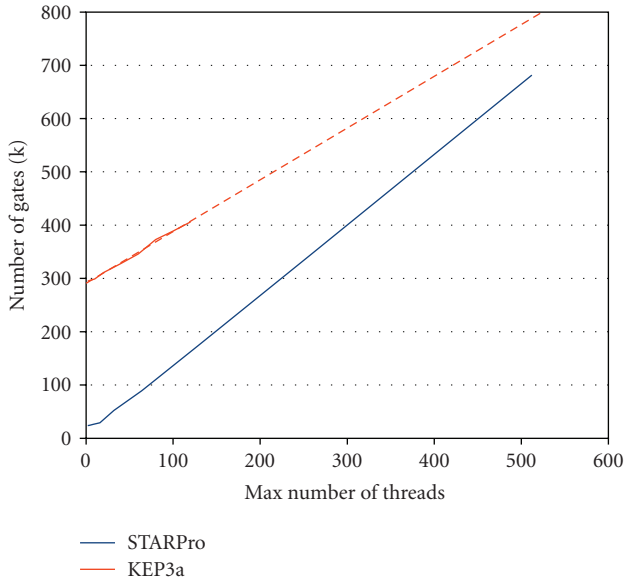


Figure 18: Comparison of hardware resource usage between KEP3a and STARPro.

performed for thread 1 between lines 45 ~ 52. This time, thread 1 breaks the barrier and revives thread 0 on lines 50 ~ 52.

Cluster C10 can only be reached when threads 1 and 2 join. The corresponding code for C10 is shown on lines 16 ~ 18. Note that the *chkabort* node positioned immediately below the *join* node. It is inserted because the *chkabort* nodes in threads 1 and 2 only cause these two threads to join. To actually exit the program, thread 0 has to take the preemption by checking for signal `R` immediately after the join.

6. Experimental Results

STARPro was successfully synthesized for both Cyclone II [20] and Spartan-3 [21] FPGAs. Its hardware resource usage on Spartan3 is presented in Table 2 for comparison with KEP3a [10]. 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 [10]. These results have been used to produce the curves in Figure 18. Both processors exhibit linear increase of required resources (logic gates) with the increase of number of threads. STARPro will not significantly change the hardware resource usage when the number of memory mapped I/O ports increases.

The benchmark programs presented here have been selected from EstBench [22]; these are listed in Table 3. All the selected programs are also present in [10] for comparison. All the Esterel examples used in the benchmark are pure control-driven and have minimal data computation. 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 STARPro pipelined architecture in terms of the execution speedup.

Table 4: The worst and average case reaction time.

Module name	KEP WCRT (clk cyc)	STARPro WCRT (clk cyc)	Speedup	KEP ACRT (clk cyc)	STARPro ACRT (clk cyc)	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

To evaluate STARPro's compiler, we compared it against four other Esterel compilers, namely, CEC v0.4 [8], EEC2 [11], the V5 [1] and V7 Esterel compilers [23]. These compilers produced C code from the Esterel source, which we compiled for the NIOS II [24] 32-bit RISC processor. NIOS is a softcore processor, provided by Altera as part of its development tools for its Cyclone II FPGA. All C programs were compiled using the nios2-elf-gcc compiler with level-2 optimization (−O2). The reason for using NiOS II was that it was implemented on the same Cyclone II FPGA as was STARPro.

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 Coverage Analysis tool produces a set of input trace that covers all possible states in the program. The worst-case reaction time is obtained from the longest reaction by feeding the generated input trace to the program.

The worst-case and average-case reaction times for KEP3a and STARPro are shown in Table 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 as to speedup in Table 4) in worst-case reaction time (WCRT), and 38% faster execution in average-case reaction time (ACRT) expressed in number of system clock cycles. We consider this comparison 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 (167 MHz versus 60 MHz) when synthesized for the same FPGA device, which gives it even further advantage in terms of real execution time. 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 size for the various compilers was 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

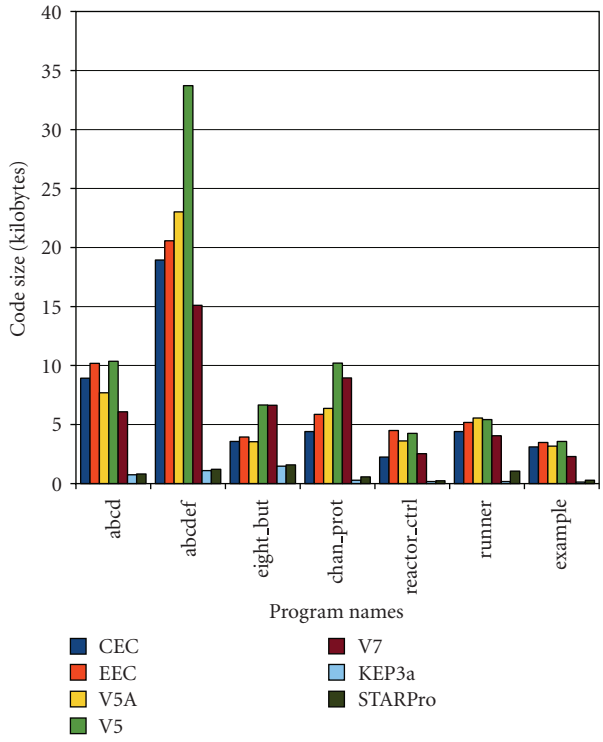


Figure 19: Code size comparison in bar graph (lower better).

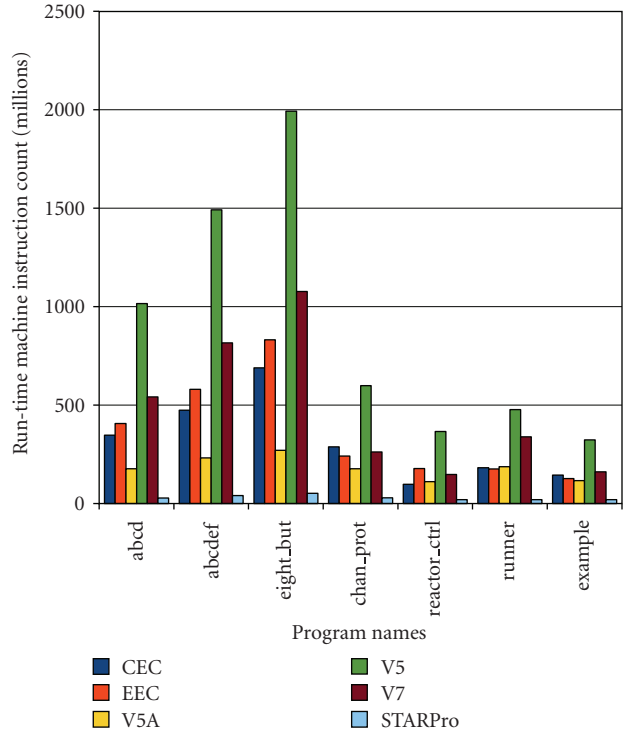


Figure 20: Performance (in run-time machine instruction count) comparison in bar graph (lower better).

Table 5: Code size comparison using different compilation techniques.

Module name	Code size (kilobytes)						
	CEC	EEC	V5A	V5	V7	KEP3a	STARPro
abcd	8.94	10.18	7.68	10.37	6.09	0.74	0.8
abcdef	18.95	20.57	23.02	33.73	15.1	1.11	1.21
eight_but	3.56	3.93	3.54	6.66	6.64	1.48	1.58
chan_prot	4.41	5.86	6.38	10.22	8.95	0.29	0.56
reactor_ctrl	2.25	4.5	3.62	4.26	2.53	0.17	0.24
runner	4.42	5.19	5.55	5.43	4.05	0.17	1.05
example	3.1	3.47	3.18	3.57	2.29	0.14	0.29

Table 6: Performance (in run-time machine instruction count).

Module name	Number of instructions (in millions)					
	Software approach using C					STARPro
	CEC	EEC	V5A	V5	V7	
abcd	347	407	177	1015	541	28
abcdef	473	580	232	1492	815	41
eight_but	689	832	270	1993	1077	51
chan_prot	288	241	176	598	261	29
reactor_ctrl	97	178	111	366	148	20
runner	181	175	186	476	338	19
example	144	127	116	323	160	20

Table 7: Effectiveness of pipelining in clock cycles per instruction.

Module name	Clk cyc	Instructions	ClkCyc/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

to the conventional software approach, as depicted in Table 5 (plotted as a graph in Figure 19). Overall, STARPro has an average 40% larger code size than KEP3a.

To compare performance of the code generated by the software compilers, we ran each Esterel program for one million reactions with randomly generated input trace. Input traces are generated once for each Esterel program. The total number of machine instructions to complete million reactions is recorded in Table 6 (plotted as graph in Figure 20).

Finally, Table 7 shows the performance gain from the pipelined STARPro architecture. 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. Multiplying the instruction count by three, we obtain the total number of clock cycles required for a nonpipelined processor. The effect of pipelining results in an average speedup of 1.83 times.

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.

7. Conclusions

We have presented a direct execution platform for Esterel with multithreading support. Esterel programs compiled for STARPro are significantly faster than those produced by Esterel software compilers for traditional processors, and at the same time have smaller memory footprint. In comparison to an existing Esterel-optimized processor, KEP3a, STARPro achieves better execution times but has larger memory footprint. This has been accomplished with a simpler hardware design, which, at the same time, consumes significantly less hardware resources. This led to the ability of the pipelined STARPro processor to operate at 167 MHz in contrast to the nonpipelined operating frequency of KEP3a of only 60 MHz for the same implementation technology.

Our future work includes further optimization of the processor itself, which will include work on hardware support for scheduling. Also, we are looking for extension of the approach to data driven computations where a standard traditional processor would be used to execute data transformations that are performed by using C functions in Esterel. Also, we are looking for the approach where multiple STARPro processors would be used as execution platform for more demanding Esterel programs.

References

[1] G. Berry, *The Esterel v5 Language Primer, Version v5 91*, Centre de Mathématiques Appliquées, Ecole des Mines, Sophia-Antipolis, France, 2000.

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

[3] G. Berry, "The constructive semantics of pure Esterel," draft book, 1999, http://www-sop.inria.fr/esterel.org.

[4] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, K. Apt, Ed., NATO ASI Series, Vol. F-13, pp. 477–498, Springer, La Colle-sur-Loup, France, 1985.

[5] K. Schneider and M. Wenz, "A new method for compiling schizophrenic synchronous programs," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*, pp. 49–58, ACM Press, New York, NY, USA, November 2001.

[6] E. Closse, 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*, vol. 65, no. 5, pp. 80–94, 2002.

[7] D. Potop-Butucaru and R. de Simone, "Optimizations for faster execution of Esterel programs," in *Proceedings of the 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pp. 227–236, Mont-Saint-Michel, France, June 2003.

[8] S. A. Edwards and J. Zeng, "Code generation in the Columbia Esterel compiler," *EURASIP Journal of Embedded Systems*, vol. 2007, Article ID 52651, 31 pages, 2007.

[9] M. W. S. Dayaratne, P. S. Roop, and Z. Salcic, "Direct execution of Esterel using reactive microprocessors," in *Proceedings of International Workshop on Synchronous Languages, Applications and Programming (SLAP '05)*, Edinburgh, Scotland, UK, April 2005.

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

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

[12] X. Li and R. von Hanxleden, "The Kiel Esterel Processor-a semicustom, configurable reactive processor," in *Proceedings of the Conference on Synchronous Programming (SYNCHRON '04)*, S. A. Edwards, N. Halbwachs, R. von Hanxleden, and T. Stauner, Eds., Dagstuhl Seminar Proceedings no. 04491, Internationales Begegnungs- und Forschungszentrum (IBFI), Dagstuhl, Germany, 2004.

[13] X. Li, 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 *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '05)*, pp. 225–236, ACM Press, San Francisco, Calif, USA, September 2005.

[14] X. Li and R. von Hanxleden, "A concurrent reactive Esterel processor based on multi-threading," in *Proceedings of the ACM Symposium on Applied Computing (SAC '06)*, pp. 912–917, Dijon, France, April 2006.

[15] S. Yuan, S. Andalam, L. H. Yoong, P. Roop, and Z. Salcic, "STARPro: a new multithreaded direct execution platform for Esterel," in *Proceedings of Model-Driven High-Level Programming of Embedded Systems (SLA++P '08)*, Budapest, Hungary, April 2008.

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

[17] Z. Salcic, D. Hui, P. S. Roop, and M. Biglari-Abhari, "ReMIC: design of a reactive embedded microprocessor core," in *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC '05)*, pp. 977–981, ACM Press, Shanghai, China, January 2005.

[18] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[19] Esterel Studio User Manual, Version 6.0, Esterel EDA Technologies SAS, Elancourt, November 2007.

[20] Altera Corp. Cyclone II FPGA, March 2009, http://www.altera.com/products/devices/cyclone2/cy2-index.jsp.

[21] Xilinx Spartan3 FPGA, March 2009, http://www.xilinx.com/products/spartan3a/index.htm.
[22] S. A. Edwards, EstBench Esterel Benchmark Suit, June 2007, http://www1.cs.columbia.edu/~sedwards/software.html.
[23] G. Berry, "The Esterel v7 Reference Manual: Version v7 30 for Esterel Studio 5.3," Esterel Technologies SA, Villeneuve-Loubet, France, December 2005.
[24] Altera Corp., November 2007, http://www.altera.com/pro-ducts/ip/processors/nios2/ni2-index.html.