Mapping Esterel onto a Multi-Threaded Embedded Processor

[Extended Abstract] *

Xin Li Marian Boldt Reinhard von Hanxleden

Real-Time and Embedded Systems Group
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{xli,mabo,rvh}@informatik.uni-kiel.de

Abstract

The synchronous language Esterel is well-suited for programming control-dominated reactive systems at the system level. It provides non-traditional control structures, in particular concurrency and various forms of preemption, which allow to concisely express reactive behavior. As these control structures cannot be mapped easily onto traditional, sequential processors, an alternative approach that has emerged recently makes use of special-purpose reactive processors. However, the designs proposed so far have limitations regarding completeness of the language support, and did not really take advantage of compile-time knowledge to optimize resource usage.

This paper presents a reactive processor, the Kiel Esterel Processor 3a (KEP3a), and its compiler. The KEP3a improves on earlier designs in several areas; most notable are the support for exception handling and the provision of context-dependent preemption handling instructions. The KEP3a compiler presented here is to our knowledge the first for multi-threaded reactive processors. The translation of Esterel's preemption constructs onto KEP3a assembler is straightforward; however, a challenge is the correct and efficient representation of Esterel's concurrency. The compiler generates code that respects data and control dependencies using the KEP3a priority-based scheduling mechanism. We present a priority assignment approach that makes use of a novel concurrent control flow graph and has a complexity that in practice tends to be linear in the size of the program. Unlike earlier Esterel compilation schemes, this approach avoids unnecessary context switches by considering each thread's actual execution state at run time. Furthermore, it avoids code replication present in other approaches.

Categories and Subject Descriptors C1.3 [*PROCESSOR ARCHITECTURES*]: Other Architecture Styles; C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Compilers, Optimization

General Terms Design, Experimentation, Languages, Performance

Keywords Reactive systems, concurrency, multi-threading, Esterel, low-power processing

1. Introduction

The programming language Esterel [6] has been designed for developing control-dominated reactive software or hardware systems. It belongs to the family of *synchronous languages*, which have a formal semantics that abstracts away run-time uncertainties, and allow abstract, well-defined and executable descriptions of the application at the system level. Hence these languages are particularly suited to the design of safety-critical real-time systems; see Benveniste *et al.* for a nice overview of synchronous languages [3]. To express reactive behavior, Esterel offers numerous powerful control flow primitives, in particular concurrency and various preemption operators. Concurrent threads can communicate back and forth instantaneously, with a tight semantics that guarantees deterministic behavior. This is valuable for the designer, but also poses implementation challenges.

In general, an Esterel program is validated via a simulation-based tool set, and then synthesized to an *intermediate language*, *e. g.*, C or VHDL. To build the real system, one typically uses a commercial off-the-shelf (COTS) processor for a software implementation, or a circuit is generated for a hardware implementation. HW/SW co-design strategies have also been investigated, for example in POLIS [2, 19].

During the past years, many techniques have been proposed to synthesize efficient software implementations from Esterel programs, typically concentrating on the generation of optimized intermediate language code (see also Section 2). However, there remain some fundamental difficulties in compiling Esterel's reactive control flow constructs to sequential, traditional processors. Reactive programs are often characterized by very frequent context switches; as our experiments indicate, a context switch after every three or four instructions is not uncommon. This adds significant overhead to the traditional compilation approaches, as the restriction to a single program counter requires the program to manually keep track of thread control counters using state variables; traditional OS context switching mechanisms would be even more expensive. Furthermore, the handling of preemptions requires a rather clumsy sequential checking of conditionals whenever control flow

* The full paper is available as a Technical Report [15]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00.

may be affected by a preemption. Hence, an alternative approach that has emerged recently makes use of special-purpose reactive processors, which strive for a direct implementation of Esterel's control flow and signal handling constructs.

In this paper, we present a reactive architecture, the Kiel Esterel Processor 3a (KEP3a), and a compiler that translates Esterel into KEP3a assembler. The development of the KEP3a was driven by the desire to achieve competitive execution speeds at minimal resource usage, considering processor size and power usage as well as instruction and data memory. A key to achieve this goal is the instruction set architecture (ISA) of the KEP3a, which allows the mapping of Esterel programs into compact machine code while still keeping the processor light-weight. Notable features of the KEP3a that go beyond earlier approaches, including the KEP3 design [17], include the following:

- Unlike earlier reactive processing approaches, the KEP3a ISA is *complete* in that it allows a direct mapping of all Esterel statements onto KEP3a assembler.
- A characteristic of the Esterel language is that its control flow operators can be combined with each other in an arbitrary fashion. This makes the language concise and facilitates formal analysis; however, it can also make unrefined processing approaches fairly costly. The KEP3a ISA therefore not only supports common Esterel statements directly, but also takes into consideration the statement context. Providing such a *refined* ISA further minimizes hardware usage while preserving the generality of the language.

The KEP3a code is typically an order of magnitude smaller than that of the MicroBlaze, a COTS RISC processor core. The worst case reaction time is typically improved by 4x, and energy consumption is also typically reduced to a quarter. Furthermore, the KEP3a is scalable to very high degrees of concurrency, increasing the maximal thread count from 2 to 120 increased the gate count by only 40%.

The rest of this paper is organized as follows. The next section discusses related work. Sections 4 and 5 present the instruction set and the architecture of the KEP3a. Section 6 discusses the compiler used to synthesize Esterel programs onto the KEP3a. Experimental results are presented in Section 7. The paper concludes in Section 8.

2. Related Work

In the past, various techniques have been developed to synthesize Esterel into software; see Edwards [11] for an overview, which also places Esterel code synthesis into the general context of compiling concurrent languages. The compiler presented here belongs to the family of simulation-based approaches, which try to emulate the control logic of the original Esterel program directly, and generally achieve compact and yet fairly efficient code. These approaches first translate an Esterel program into some specific graph formalism that represents computations and dependencies, and then generate code that schedules computations accordingly. The EC/Synopsys compiler first constructs a *concurrent control flow graph* (CCFG), which it then sequentializes [10]. Threads are statically interleaved according to signal dependencies, with the potential drawback of superfluous context switches; furthermore, code sections may be duplicated if they are reachable from different control points ("surface"/"depth" replication [4]). The SAXO-RT compiler [8] divides the Esterel program into basic blocks, which schedule each other within the current and subsequent logical tick. An advantage relative to the Synopsis compiler is that it does not perform unnecessary context switches and largely avoids code duplications; however, the scheduler it employs has an overhead proportional to the total number of basic blocks present in the program. The grc2c compiler [21] is based on the *graph code* (GRC) format, which preserves the state-structure of the given program and uses static analysis techniques to determine redundancies in the activation patterns. A variant of the GRC has also been used in the *Columbia Esterel Compiler* (CEC) [12], which again follows SAXO-RT's approach of dividing the Esterel program into atomically executed basic blocks. However, their scheduler does not traverse a score board that keeps track of all basic blocks, but instead uses a compact encoding based on linked lists, which has an overhead proportional to just the number of blocks actually executed.

In summary, there is currently not a single Esterel compiler that produces the best code on all benchmarks, and there is certainly still room for improvements. For example, the simulation-based approaches presented so far restrict themselves to interleaved single-pass thread execution, which in the case of repeated computations ("schizophrenia" [4]) requires code replications; it should be possible to avoid this with a more flexible scheduling mechanism.

Driven by these limitations of traditional processors, the *reactive processing* approach tries to achieve a more efficient execution of reactive programs by providing an ISA that is a better match for reactive programming. The architectures proposed so far specifically support Esterel programming; however, they should be an attractive alternative to traditional processor architectures for reactive programming in general. Two strategies have been proposed to implement the reactive processing approach, which have been classified as the patched reactive processor approach and the custom reactive processor approach [16]. The *patched reactive processor* strategy combines a COTS processor core with an external hardware block, which implements additional Esterel-style instructions. To our knowledge, the ReFLIX and RePIC processors [22] were the first processors of this type. Based on RePIC, Dayaratne *et al.* proposed an extension to a multi-processor architecture, the EMPEROR [23], which allows the distributed execution of Esterel programs and also handles Esterel's concurrency operator. The EMPEROR uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential for execution speed-ups relative to single-processor implementations. However, their execution model potentially requires to replicate parts of the control logic at each processor. The EMPEROR Esterel Compiler 2 (EEC2) [23] is based on a variant of the GRC, and appears to be competitive even for sequential executions on a traditional processor. However, their synchronization mechanism, which is based on a three-valued signal logic, does not seem able to take compile-time scheduling knowledge into account, and instead repeatedly cycles through all threads until all signal values have been determined.

The *custom reactive processor* strategy consists of a full-custom reactive core with an instruction set and data path tailored for processing Esterel code. The Kiel Esterel Processor (KEP) family follows this route. The KEP2 provides preemption primitives, further Esterel constructs such as valued signals, signal counters, and the pre operator, and a Tick Manager that allows to run the KEP at a constant logical tick rate and detects timing overruns [16]. A compiler for the KEP2, based on the *KEP assembler graph* (KAG), also performs a Worst Case Reaction Time (WCRT) analysis, which determines the maximal number of instruction cycles executed within a logical tick. However, neither the KEP2 nor the compiler for it support Esterel's concurrency operator. The compiler presented here uses a concurrent extension of the KAG, the *Concurrent KEP Assembler Graph* (CKAG).

The basic design of the KEP3a processor follows that of the KEP3 [17], which employs a *Thread Block* to handle concurrency via multi-threading. The KEP3 ISA is efficient in that it most commonly used Esterel statements can be expressed directly with just a single KEP3a instruction. However, the KEP3 still has several weaknesses, which have been overcome in the KEP3a. In particular, the KEP3a is incomplete in its support of Esterel kernel statements in that it does not support exception handling. Furthermore, the KEP3 is unrefined in its preemption handling, as it did not consider the execution context when mapping preemption statements to hardware units. Finally, no compilation scheme has been presented so far for the multi-threaded execution model of the KEP3.

3. The Esterel Language

The execution of an Esterel program is divided into logical *instants*, or *ticks*, and communication within or across threads occurs via *signals*; at each tick, a signal is either *present* (*emitted*) or *absent* (not emitted). Esterel statements are either *transient*, in which case they do not consume logical time, or *delayed*, in which case execution is finished for the current tick. Per default statements are transient, and these include for example emit, loop, present, or the preemption operators; delayed statements include pause, (non-immediate) await, and every.

Esterel's parallel operator, ||, groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated.

Esterel offers two types of preemption constructs: An *abortion* kills its body when a delay elapses. We distinguish *strong* abortion, which kills its body immediately (at the beginning of a tick), and *weak* abortion, which lets its body receive control for a last time (abortion at the end of the tick). A *suspension* freezes the state of a body in the instant when the trigger event occurs.

Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are complications when traps are nested or when the trap scope includes concurrent threads. The following rules apply: if one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent threads are weakly aborted; if concurrent threads execute multiple exit instructions in the same tick, the outermost trap takes priority.

3.1 An Example

As an example, consider the Esterel program Edwards02 [10, 8], shown in Figure 1(a). This program implements the following behavior: whenever the signal S is present, (re-)start two concurrent threads. The first thread first awaits a signal I; it then continuously emits R until A is present, in which case it emits R one last time (weak abortion of the sustain), emits O, and terminates. The second thread tests every other tick for the presence of R, in which case it emits A.

3.2 Statement Dismantling

At the Esterel level, one distinguishes *kernel statements* and *derived statements*; the derived statements are basically syntactic sugar, built up from the kernel statements. In principle, any set of Esterel statements from which the remaining statements can be constructed can be considered a valid set of kernel statements, and the accepted set of Esterel kernel statements has evolved over time. For example, the halt statement used to be considered a kernel statement, but is now considered to be derived from loop and pause. We here adopt the definition of which statements are kernel statements from the v5 standard [5]. The process of expanding derived statements into equivalent, more primitive statements—which may or may not be kernel statements—is also called *dismantling*. The Esterel program Edwards02-dism [10], shown in Figure 1(b), is a dismantled version of the program Edwards02. It is instructive to compare this program to the original, undismantled version.

3.3 Dependency Cycles

A consequence of Esterel's synchronous model of execution is that there may be *dependency cycles*, which involve concurrent threads communicating back and forth within one tick. Such dependency cycles must be *broken*, for example by a delay node, because otherwise it would not be possible for the compiler to devise a valid execution schedule that obeys all ordering constraints. In the Edwards02 example, there is one dependency cycle, from the sustain R_9 instruction[1] in the first parallel thread to the present R_{16} in the second parallel to the emit A_{17} back to the sustain R_9, which is weakly aborted whenever A is present. The dependency cycle is broken in the dismantled version, as there the sustain R has been separated into signal emission (emit R_{22}) and a delay (pause$_{23}$, enclosed in a loop. The broken dependency cycle can also be observed in the CKAG, shown in Figure 1(c). Referring to CKAG nodes by the corresponding line numbers (the "Lxx" part of the node labels), the cycle is L14 → L23 → L24 → L17 → L18 → L14; it is broken by the delay in L17. The CKAG is explained in more detail in Section 6.1.

4. The KEP3a Instruction Set Architecture

When designing an instruction set architecture to implement Esterel-like programs, it would in principle suffice to just implement the kernel statements—plus some additions that go beyond "pure" Esterel, such as valued signals, local registers, and support for complex signal and data expressions. However, we decided against that, in favor of an approach that includes some redundancy among the instructions to allow more compact and efficient object code.

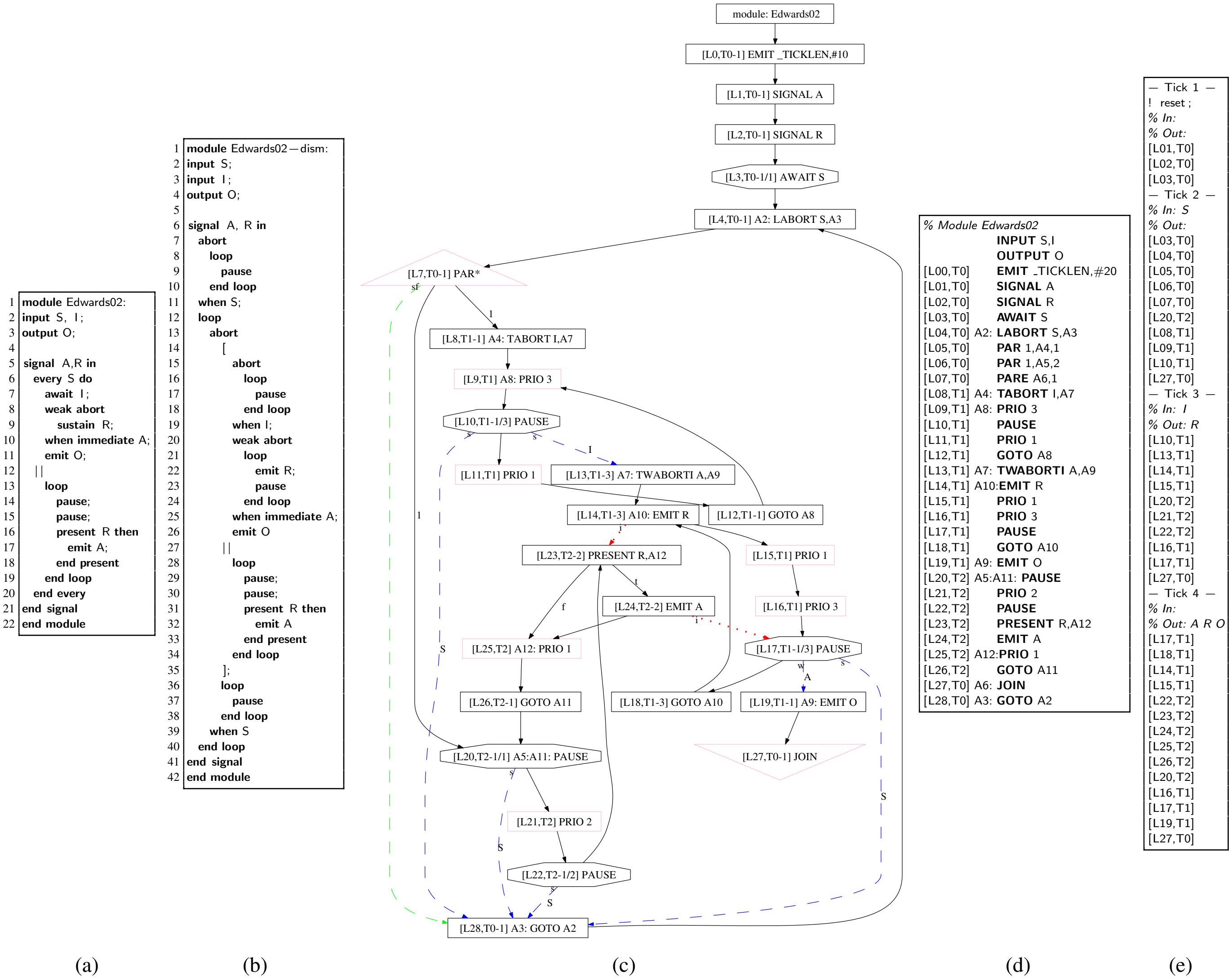
The resulting KEP3a ISA is summarized in Table 1, which also illustrates the relationship between Esterel statements and the KEP3a instructions. Not shown are some classical arithmetic and logic operations and control flow primitives that the KEP3a provides to implement the complete range of Esterel's signal and data expressions [15].

The KEP3a uses a 36-bit wide instruction word and a 32-bit data bus. The KEP3a ISA has the following characteristics:

- All the kernel Esterel statements, and some frequently used derived statements, can be mapped to KEP3a instructions directly. For the remaining Esterel statements there exist direct expansion rules that allow the compiler to still generate KEP3a code, including general signal expressions.
- Common Esterel expressions, in particular all of the delay expressions (*i. e.*, standard, immediate, and count delays), can be represented directly.
- The control statements are fully orthogonal, their behavior matches the Esterel semantics in all execution contexts.
- Valued signals and other signal expressions, *e. g.*, the previous value of a signal and the previous status of a signal, are also directly supported.
- All instructions fit into one instruction word and can be executed in a single instruction cycle, except for instructions that contain count delay expressions, which need an extra instruction word and take another instruction cycle to execute.

The KEP3a also handles schizophrenic programs [4] correctly—if an Esterel statement must be executed multiple times within a tick, the KEP3a simply does so [15]. The SIGNAL instruction also ensures that reincarnated signals are correctly initialized.

[1] To aid readability, we here use the convention of subscripting instructions with the line number where they occur.



```
module Edwards02:
input S, I;
output O;

signal A,R in
  every S do
    await I;
    weak abort
      sustain R;
    when immediate A;
    emit O;
  ||
    loop
      pause;
      pause;
      present R then
        emit A;
      end present
    end loop
  end every
end signal
end module
```

(a)

```
module Edwards02—dism:
input S;
input I;
output O;

signal A, R in
  abort
    loop
      pause
    end loop
  when S;
  loop
    abort
      [
        abort
          loop
            pause
          end loop
        when I;
        weak abort
          loop
            emit R;
            pause
          end loop
        when immediate A;
        emit O
      ||
        loop
          pause;
          pause;
          present R then
            emit A
          end present
        end loop
      ];
      loop
        pause
      end loop
    when S
  end loop
end signal
end module
```

(b)

(c)

```
% Module Edwards02
                INPUT S,I
                OUTPUT O
[L00,T0]        EMIT _TICKLEN,#20
[L01,T0]        SIGNAL A
[L02,T0]        SIGNAL R
[L03,T0]        AWAIT S
[L04,T0] A2:    LABORT S,A3
[L05,T0]        PAR 1,A4,1
[L06,T0]        PAR 1,A5,2
[L07,T0]        PARE A6,1
[L08,T1] A4:    TABORT I,A7
[L09,T1] A8:    PRIO 3
[L10,T1]        PAUSE
[L11,T1]        PRIO 1
[L12,T1]        GOTO A8
[L13,T1] A7:    TWABORTI A,A9
[L14,T1] A10:EMIT R
[L15,T1]        PRIO 1
[L16,T1]        PRIO 3
[L17,T1]        PAUSE
[L18,T1]        GOTO A10
[L19,T1] A9:    EMIT O
[L20,T2] A5:A11: PAUSE
[L21,T2]        PRIO 2
[L22,T2]        PAUSE
[L23,T2]        PRESENT R,A12
[L24,T2]        EMIT A
[L25,T2] A12:PRIO 1
[L26,T2]        GOTO A11
[L27,T0] A6:    JOIN
[L28,T0] A3:    GOTO A2
```

(d)

```
— Tick 1 —
! reset ;
% In:
% Out:
[L01,T0]
[L02,T0]
[L03,T0]
— Tick 2 —
% In: S
% Out:
[L03,T0]
[L04,T0]
[L05,T0]
[L06,T0]
[L07,T0]
[L20,T2]
[L08,T1]
[L09,T1]
[L10,T1]
[L27,T0]
— Tick 3 —
% In: I
% Out: R
[L10,T1]
[L13,T1]
[L14,T1]
[L15,T1]
[L20,T2]
[L21,T2]
[L22,T2]
[L16,T1]
[L17,T1]
[L27,T0]
— Tick 4 —
% In:
% Out: A R O
[L17,T1]
[L18,T1]
[L14,T1]
[L15,T1]
[L22,T2]
[L23,T2]
[L24,T2]
[L25,T2]
[L26,T2]
[L20,T2]
[L16,T1]
[L17,T1]
[L19,T1]
[L27,T0]
```

(e)

Figure 1. The Edwards02 example [10]: (a) Esterel; (b) Esterel after dismantling; (c) Concurrent KEP Assembler Graph (CKAG, see Section 6.1), where rectangles are transient nodes, octagons are delay nodes, and triangles are fork/join nodes; (d) KEP assembler, (e) trace of execution. The CKAG nodes and the KEP assembler include labels (in brackets) that list the line number ("Lxx") and thread id ("Tx"). The CKAG labels also include "-$prio$[/$prionext$]" as appropriate. CKAG control successor edges are solid, other successor edges are dashed, dependency edges are dotted; tail labels further indicate the edge type.

4.1 The Example

The KEP3a assembler code for the Edwards02 example is shown in Figure 1(d). At the beginning of a module, an EMIT _TICKLEN$_{L00}$ instruction assigns the Tick Manager a certain value to define an upper bound on the number of instruction cycles within a logical tick [16]. The following SIGNAL instructions initialize local signals. The LABORT S,A3$_{L04}$ configures a local watcher to perform an abort of the abort block delimited by the label A3 whenever S is present; preemptions are discussed further in Section 5.2. The subsequent PAR/PARE instructions fork off the two parallel threads, which are joined in L27; this is explained in Section 5.1. Overall, while some of the details are still missing, one should already see a fairly close correspondence between the assembler code and the dismantled Esterel version. The total number of assembler instructions is between the line count of the original Esterel program and its dismantled counterpart, which indicates a very compact encoding of the program.

A possible execution trace for this example is shown in Figure 1(e). In the first tick, the main thread (T0) executes three instructions, and then pauses at the AWAIT S$_{L03}$. No input signals are present, and none are emitted. In the second tick, the input signal S is present, hence the main thread passes the AWAIT S$_{L03}$, enters the abort scope for S, and forks off the parallel threads (L05–L07). The parallel thread T2 then executes a PAUSE$_{L20}$, followed by context switch to T1, which executes another three instructions. Finally, the main thread executes JOIN$_{L27}$, which tests whether the threads can be joined already, *i. e.*, whether they both have already terminated—which so far is not the case. Similarly, the trace shows the interleaved execution of the parallel threads for ticks 3 and 4, which again is explained in more detail in Section 5.1.

Mnemonic, Operands	**Esterel Syntax**	**Notes**
PAR *Prio*, *startAddr* [, *ID*] PARE *endAddr* JOIN	[*p* || *q*]	Fork and join, see also Section 5.1. An optional *ID* explicitly specifies the ID of the created thread.
PRIO *Prio*		Set the priority of the current thread.
[L|T][W]ABORT [*n*,] *S*, *endAddr*	[weak] abort ... when [*n*] *S*	The prefix [L|T] denotes the type of watcher to use, see also Section 5.2. **L** : Local Watcher **T** : Thread Watcher ***none*** : general Watcher
[L|T][W]ABORTI *Sexp*, *endAddr*	[weak] abort ... when immediate *Sexp*	
SUSPEND[I] *Sexp*, *endAddr*	**suspend** ... **when** [immediate] *Sexp*	
EXIT *addr*	**trap** *T* **in** **exit** *T* **end trap**	Exit from a trap, *addr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ||.
PAUSE	**pause**	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [*n*,] *Sexp*	await [*n*] *Sexp*	
AWAIT[I] *Sexp*	await [immediate] *Sexp*	

CAWAITS
CAWAIT[I] S, *addr*
CAWAITE
	await
	case [immediate] *Sexp* do
	end
		Wait for several signals in parallel.

SIGNAL *S*	**signal** *S* **in** ... **end**	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	sustain *S* [(*val*)]	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	Jump to *elseAddr* if *S* is absent.
NOTHING	**nothing**	Do nothing.
HALT	halt	Halt the program.
GOTO *addr*	**loop** ... **end loop**	Jump to *addr*.

Table 1. Overview of the KEP3a Esterel-type instruction set architecture. Esterel kernel statements are shown in **bold**. A signal expression *Sexp* can be one of the following: 1. *S*: signal status (present/absent); 2. PRE(*S*): previous status of signal: 3. TICK: always present. A numeral *n* can be one of the following: 1. #*data*: immediate data; 2. *reg*: register contents; 3. ?*S*: value of a signal; 4. PRE(?*S*): previous value of a signal.

5. The KEP3a Processor Architecture

The main challenge when designing a reactive architecture is the handling of control. In the KEP3a, the Reactive Multi-Threading Core (RMC) is responsible for managing the control flow of all threads. Figure 2 shows the architecture of the RMC. It contains dedicated hardware units to handle concurrency, preemption, exceptions, and delays. In the following, we will briefly discuss each of these in turn.

5.1 Handling Concurrency

To implement concurrency, the KEP3a employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses and dynamically changing priorities. The scheduler is very light-weight. In the KEP3a, scheduling and context switching do not cost extra instruction cycles, only changing a thread's priority costs an instruction. The priority-based execution scheme allows on the one hand to enforce an ordering among threads that obeys the constraints given by Esterel's semantics, but on the other hand avoids unnecessary context switches. If a thread lowers its priority during execution but still has the highest priority, it simply keeps executing.

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into KEP assembler as follows [17]. First, threads are *forked* by a series of instructions that consist of n PAR instructions and one PARE instruction. Each PAR instruction creates one thread, by assigning a non-negative *priority* and a *start address*. The end address of the thread is either given implicitly by the start address specified in a subsequent PAR instruction, or, if there is no more thread to be created, it is specified in a PARE instruction. The code block for the last thread is followed by a JOIN instruction, which waits for the termination of all forked threads and concludes the concurrent statement. The example in Figure 1(d) illustrates this; instructions L08–L19 constitute thread T1, thread T2 spans L20–L26, and the remaining instructions belong to the main thread, T0.

The priority of a thread is assigned when the thread is created (with the aforementioned PAR instruction), and can be changed subsequently by executing a priority setting instruction (PRIO). A threads keeps its priority across delay instructions; that is, at the start of a tick it resumes execution with the priority it had at the end of the previous tick.

When a concurrent statement terminates, through regular termination of all concurrent threads or via an exception/abort, the priorities associated with the terminated threads also disappear, and the priority of the main thread is restored to the priority upon entering the concurrent statement.

The execution status of a thread is illustrated in Figure 3, using the SyncChart formalism [1]. Two flags are needed to describe the status of a thread. One flag indicates whether the thread is *disabled* or *enabled*. Initially, only the main thread (T0) is enabled. Other threads become enabled whenever they are forked, and become disabled again when they are joined after finishing all statements in their body, or when the preemption control tries to set its program

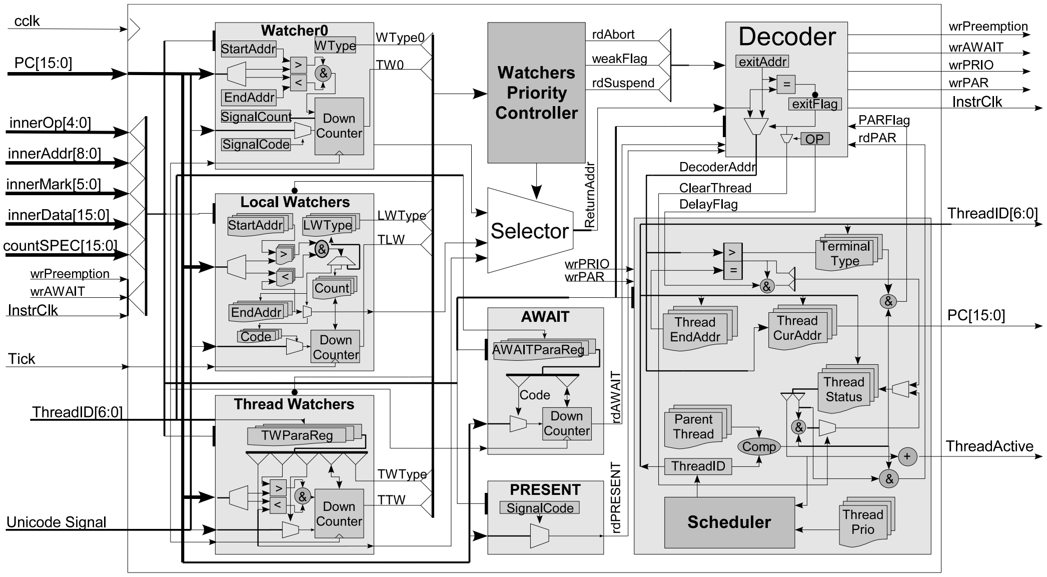


Figure 2. The architecture of the KEP3a Reactive Multi-threaded Core (RMC).

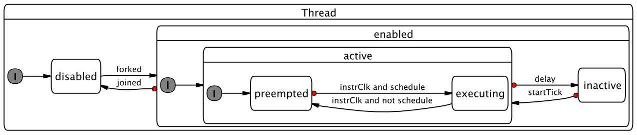


Figure 3. Execution status of a single thread.

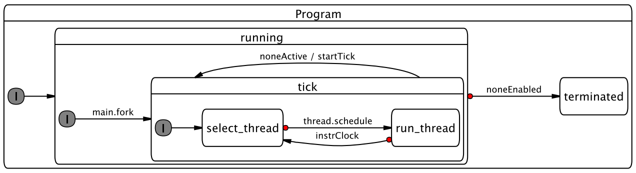


Figure 4. The status of the whole program, as managed by the Thread Block.

counter to a value which is out of the thread address range. The other flag indicates whether the thread should still be scheduled within the current logical tick (the thread is *active*) or not (*inactive*).

The Thread Block is responsible for managing threads, as illustrated in Figure 4. Upon program start, the main thread is enabled (forked), and the program is considered *running*. Subsequently, for each instruction cycle, the Thread Block decides which thread ought to be scheduled for execution in this instruction cycle. The thread which has the highest priority among all active threads will be selected for execution and becomes *executing*. The main thread always has priority 1, which is the lowest possible priority. If there are multiple threads that have highest priority, the thread with the highest id is scheduled. Threads that are active but not executing are considered *preempted*. If there are still enabled threads, but none is active anymore, the next tick is started. If no threads are enabled anymore, the whole program is *terminated*.

Consider Tick 4 of the execution trace in Figure 1(e). At the start of the tick, all threads are active. Thread T1 has priority 3, because of the PRIO 3_{L16} instruction executed in the previous tick, and similarly T2 has priority 2. Hence, thread T1 is scheduled first and keeps running until it executes PRIO 1_{L15}, which causes its priority to become lower than T2's. Thus there is a context switch to T2, which runs until it executes $\mathsf{PAUSE}_{\mathsf{L20}}$, where it finishes its tick and becomes inactive, and the still active T1 becomes running again. As can be seen from this trace, all scheduling constraints present in the original Esterel program are nicely obeyed by the interleaved execution.

The control of a thread can never exceed its address range, and if a thread still tries to do so, it will be terminated immediately. This mechanism gives a neat solution for handling arbitrary preemption and concurrency nests. For example, assume a strong abortion, which nests several threads, is triggered: the abortion will cause each thread to try to jump to the end of the abortion block, which will be beyond its address range, and hence the thread will be terminated.

5.2 Handling Preemption

The RMC provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body [17]. When preemptions are nested and triggered simultaneously, the Watcher Priority Controller decides which must take precedence. The KEP3 Watchers are designed to permit arbitrary nesting of preemptions, and also the combination with the concurrency operator. However, in practice this often turns out to be more general than necessary, and hence wasteful of hardware resources. Therefore, the KEP3a also includes trimmed-down versions of the Watcher. The least powerful, but also cheapest variant is the Thread Watcher, which belongs to a thread directly, and can neither include concurrent threads nor other preemptions. An intermediate variant is the Local Watcher, which may include concurrent threads and also preemptions handled by a Thread Watcher, but cannot include another Local Watcher. In the KEP3a Edwards02 code in Figure 1(d), there are three preemptions (lines L04, L08, L13), which could all be mapped to a full-size Watcher; however, this is not needed in any of these preemptions. In this example, the first preemption requires a Local Watcher, and the others can be handled with Thread Watchers.

5.3 Handling Exceptions

The KEP3a does not need an explicit equivalent to the trap statement, but it provides an EXIT statement. If a thread executes an EXIT instruction, it tries to perform a jump to the end of the trap scope. If that address is beyond the range of the current thread, control is not transferred directly to the end of the trap scope, but instead to the JOIN instruction at the end of the current thread. If other threads that merge at this JOIN are still active, they will still be allowed to execute within the current logical tick. It may be the case that a concurrent thread executes another EXIT, in which case the exception handler must decide which exception should take priority, based on the corresponding trap scopes. Once all joining threads have completed within the current tick, control is transferred to the end of the trap scope—unless there is another intermediate JOIN instruction. This process continues until control has reached the thread that has declared the trap.

5.4 Handling Delays

Delay expressions are used in temporal statements, *e. g.*, await or abort. There are three forms of delay expressions, *i. e.*, standard delays, immediate delays, and count delays. A delay may elapse in some later instant, or in the same instant in the case of immediate delays. In the KEP3a, the await statement is implemented via the AWAIT component. Every thread has its own await-component to store the parameters of the await-type statement, *e. g.*, the value of count delays. For the preemption statements, every Watcher (including its trimmed-down derivatives) also has an independent counter to handle the delays. The KEP3a can handle all delay expressions directly and exactly.

6. The KEP3a Compiler

We have implemented a compiler for the KEP3a based on the CEC infrastructure [9]. A central problem for compiling Esterel onto the KEP is the need to manage thread priorities during their creation and their further execution. In the KEP setting, this is not merely a question of efficiency, but a question of correct execution.

The priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. Due to the non-linear control flow, it is still possible that a given statement may be executed with varying priorities; in principle, the architecture would therefore allow a fully dynamic scheduling. However, we here assume that the given Esterel program can be executed with a statically determined schedule, which requires that there are no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [13] and the CEC [9] compilers; see also Section 6.1. Note that there are also Esterel programs that are causally correct (*constructive* [6]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP3a assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [18], which can then be translated into KEP3a assembler. Hence, the actual runtime schedule of a concurrent program running on KEP3a is *static* in the sense that if two statements that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each statement is known in advance. However, the run-time schedule is *dynamic* in the sense that due to the non-linear control flow and the independent advancement of each program counter, it in general cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions; a run-time scheduling mechanism is needed that manages the interleaving according to the priority and actual program counter of each active thread.

To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each tick, all enabled threads are activated, and are subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction, in which case it keeps that new priority until it executes yet another PRIO instruction, and so on. Neither the scheduler nor other threads can change its priority. Note also that a PRIO instruction is considered instantaneous. The only non-instantaneous instructions, which delimit the logical ticks and are also referred to *delayed* instructions, are the PAUSE instruction and derived instructions, such as AWAIT and SUSTAIN. This mechanism has a couple of implications:

- At the start of a tick, a thread is resumed with the priority corresponding to the last PRIO instruction it executed during the preceding ticks, or with the priority it has been created with if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it is not sufficient to execute a PRIO instruction at the beginning of that tick; instead, we must already have executed that PRIO instruction in the preceding tick.
- A thread is executed only if no other active thread has a higher priority. Once a thread is executing, it continues until a delayed statement is reached, or until its priority is lower than that of another active thread or equal to that of another thread with higher id. While a thread is executing, it is not possible for other inactive threads to become active; furthermore, while a thread is executing, it is not possible for other threads to change their priority. Hence, the only way for a thread's priority to become lower than that of other active threads is to execute a PRIO instruction that lowers its priority below that of other active threads.

As an intermediate representation for the compilation of Esterel to KEP3a, including the thread priority assignment, we use a directed graph structure called Concurrent KEP Assembler Graph (CKAG), discussed in the next section.

6.1 The Concurrent KEP Assembler Graph (CKAG)

The CKAG is generated from the Esterel program via a simple structural translation. The only non-trivial aspect is the determination of non-instantaneous paths [15], which is needed for certain edge types. Also, for convenience, we label nodes with KEP3a instructions; however, we could alternatively have used Esterel instructions as well.

The CKAG distinguishes the following sets of nodes:

***D*:** Delay nodes (octagons), which correspond to delayed KEP instructions (PAUSE, AWAIT, HALT, SUSTAIN), shown as octagons.

***F*:** Fork nodes (triangles), corresponding to PAR/PARE, shown as triangles.

***T*:** Transient nodes. This includes EMIT, PRESENT, etc., shown as rectangles, and JOIN nodes, shown as inverted triangles, but excludes fork nodes.

***N*:** Set of all nodes, $N = D \cup F \cup T$.

For each fork node n we define

***n.join*:** the JOIN statement corresponding to n,

***n.sub*:** the transitive closure of nodes in threads generated by n.

For abort nodes n ([L|T][W]ABORT[I], SUSPEND[I]) we define

$n.end$: the end of the abort scope opened by n,

$n.scope$: the nodes within n's abort scope.

For all nodes n we define

$n.prio$: the priority that the thread executing n should be running with.

For $n \in D \cup F$, we also define

$n.prionext$: the priority that the thread executing n should be resumed with in the subsequent tick.

It turns out that analogously to the distinction between $prio$ and $prionext$, we must distinguish between dependencies that affect the current tick and the next tick:

$n.dep_i$: the dependency sinks with respect to n at the current tick (the *immediate dependencies*),

$n.dep_d$: the dependency sinks with respect to n at the next tick (the *delayed dependencies*).

In general, dependencies are immediate. An exception are dependencies between emissions of strong abort trigger signals and corresponding delay nodes within the abort scope, because then the signal emission affects the behavior of the delay node not at the tick when it is entered (at the end of a tick), but at the tick when it is restarted (at the beginning of a tick).

A non-trivial task when defining the CKAG structure is to properly distinguish the different types of possible control flow, in particular with respect to their timing properties (instantaneous or delayed). We define the following types of successors for each n:

$n.suc_c$: the set of nodes that follow sequentially after n (the *control successors*).

For $n \in F$, $n.suc_c$ includes the nodes corresponding to the beginnings of the forked threads. If n is the last node of a concurrent thread, $n.suc_c$ includes the node for the corresponding JOIN—unless n's thread is instantaneous and has a (provably) non-instantaneous sibling thread.

Furthermore, the control successors exclude those reached via a preemption ($n.suc_w, n.suc_s$)—unless n is an immediate strong abortion node, in which case $n.end \in n.suc_c$.

$n.suc_w$: if $n \in D$, this is the set of nodes to which control can be transferred immediately, that is when entering n at the end of a tick, from n to via an abort; if n exits a trap, then $n.suc_w$ contains the end of the trap scope; otherwise $\emptyset$ (the *weak abort successors*).

If $n \in D$ and $n \in m.scope$ for some abort node m, it is $m.end \in n.suc_w$ in case of a weak immediate abort, or in case of a weak abort if there can (possibly) be a delay between m and n.

$n.suc_s$: if $n \in D$, this is the set of nodes to which control can be transferred after a delay, that is when restarting n at the beginning of a tick, from n to via an abort; otherwise $\emptyset$ (the *strong abort successors*).

If $n \in D$ and $n \in m.scope$ for some strong abort node m, it is $m.end \in n.suc_s$.

Note that this is not a delayed abort in the sense that an abort signal in one tick triggers the preemption in the next tick. Instead, this means that first a delay has to elapse, and the abort signal must be present at the next tick (relative to the tick when n is entered) for the preemption to take place.

$n.suc_f$: the set $n.suc_c \cup n.suc_w \cup n.suc_s$ (the *flow successors*).

For $n \in F$ we also define the following *fork abort successors*, which serve to ensure a correct priority assignment to parent threads in case there is an abort out of a concurrent statement:

$n.suc_{wf}$: the union of $m.suc_w \setminus n.sub$ for all $m \in n.sub$ where there exists an instantaneous path from n to m (the *weak fork abort successors*).

$n.suc_{sf}$: the set $\cup\{(m.suc_w \cup m.suc_s) \setminus n.sub \mid m \in n.sub\} \setminus n.suc_{wf}$ (the *strong fork abort successors*).

As already mentioned, we assume that the given program does not have cycles. However, what exactly constitutes a cycle in an Esterel program is not obvious, and to our knowledge there is no commonly accepted definition of cyclicity at the language level. The Esterel compilers that require acyclic programs differ in the programs they accept as "acyclic" (for example, the CEC accepts some programs that the v5 compiler rejects and vice versa [18]), and a full discussion of this issue goes beyond the scope of this paper. We want to consider a program cyclic if the priority assignment algorithm presented in the next section fails. This results in the following definition, based on the CKAG.

Definition 1. ***(Program Cycle)*** *An Esterel program is* cyclic *if the corresponding CKAG contains a path from a node to itself, where for all nodes n and their successors along that path, n' and n'', the following holds:*

$$\begin{array}{ll} & n \in D \wedge n' \in n.suc_w \\ \vee & n \in F \wedge n' \in n.suc_c \cup n.suc_{wf} \\ \vee & n \in T \wedge n' \in n.suc_c \cup n.dep_i \\ \vee & n \in T \wedge n' \in n.dep_d \wedge n'' \in n'.suc_c \cup n'.suc_s \cup n'.suc_{sf}\,. \end{array}$$

6.2 Computing Thread Priorities

The task of the priority algorithm is to compute a priority assignment that respects the Esterel semantics as well as the execution model of the KEP3a. The algorithm computes for each reachable node n in the CKAG the priority $n.prio$ and, for nodes in $D \cup F$, $n.prionext$. According to the Esterel semantics and the observations made in Section 6.1, a correct priority assignment must fulfill the following constraints, where m, n are arbitrary nodes in the CKAG.

Constraint 1 (Dependencies)**.** *A thread executing a dependency source node must have a higher priority than the corresponding sink. Hence, for $m \in n.dep_i$, it must be $n.prio > m.prio$, and for $m \in n.dep_d$, it must be $n.prio > m.prionext$.*

Constraint 2 (Intra-Tick Priority)**.** *Within a logical tick, a thread's priority cannot increase. Hence, for $n \in D$ and $m \in n.suc_w$, or $n \in F$ and $m \in n.suc_c \cup n.suc_{wf}$, or $n \in T$ and $m \in n.suc_c$, it must be $n.prio \geq m.prio$.*

Constraint 3 (Inter-Tick Priority for Delay Nodes)**.** *To ensure that a thread resumes computation from some delay node n with the correct priority, $n.prionext \geq m.prio$ must hold for all $m \in n.suc_c \cup n.suc_s$.*

Constraint 4 (Inter-Tick Priority for Fork Nodes)**.** *To ensure that a main thread that has executed a fork node n resumes computation—after termination of the forked threads—with the correct priority, $n.prionext \geq n.join.prio$ must hold. Furthermore, $n.prionext \geq m.prio$ must hold for all $m \in n.suc_{sf}$.*

The algorithm to assign priorities is shown in Figure 5. The algorithm starts in main(), which, after some initializations, in line 8 calls getPrio() for all nodes that must yet be processed. This set of nodes, given by $N_{ToDo} \setminus V_{prio}$ (for "Visited"), initially just contains the root of the CKAG. After $prio$ has been computed for

```
1  procedure main()
2    forall $n \in N$ do
3      $n.prio := -1$
4    $V_{prio} := \emptyset$
5    $V_{prionext} := \emptyset$
6    $N_{ToDo} := n_{root}$
7    while $\exists n \in N_{ToDo} \setminus V_{prio}$ do
8      getPrio($n$)
9    forall $n \in ((D \cup F) \cap V_{prio})$
10       $\setminus V_{prionext}$ do
11     getPrioNext($n$)
12 end
```

```
1  function getPrioNext($n$)
2    if $n.prionext = -1$ then
3      if $(n \in V_{prionext})$ then
4        error ("Cycle detected!")
5      $V_{prionext} \cup= n$
6      if $n \in D$ then
7        $n.prionext :=$
8            prioMax($n.suc_c \cup n.suc_s$)
9      elseif $n \in F$ then
10       $n.prionext :=$
11           max($n.join.prio$,
12               prioMax($n.suc_{sf}$))
13     end
14   end
15   return $n.prionext$
16 end
```

```
1  function prio [Next]Max($M$)
2    $p := 0$
3    forall $n \in M$ do
4      $p :=$ max($p$, getPrio[Next]($n$))
5    return $p$
6  end
```

```
1  function getPrio($n$)
2    if $n.prio = -1$ then
3      if $(n \in V_{prio})$ then
4        error ("Cycle detected!")
5      $V_{prio} \cup= n$
6      if $n \in D$ then
7        $n.prio :=$ prioMax($n.suc_w$)
8        $N_{ToDo} \cup= n.suc_c \cup n.suc_s$
9      elseif $n \in F$ then
10       $n.prio :=$ prioMax(
11           $n.suc_c \cup n.suc_{wf}$)
12       $N_{ToDo} \cup= n.suc_{sf} \cup$
13           $n.join.prio$
14     elseif $n \in T$ then
15       $n.prio :=$ max(
16         prioMax($n.suc_c$),
17         prioMax($n.dep_i$) + 1,
18         prioNextMax($n.dep_d$) + 1)
19     end
20   end
21   return $n.prio$
22 end
```

Figure 5. Algorithm to compute priorities.

all reachable nodes in the CKAG, a forall loop computes *prionext* for reachable delay/fork nodes that have not been computed yet.

getPrio() first checks whether it has already computed *n.prio*. If not, it then checks for a recursive call to itself (lines 3/4, see also Lemma 1). The remainder of getPrio() computes *n.prio* and, in case of delay and fork nodes, adds nodes to the N_{ToDo} list. Similarly getPrioNext() computes *n.prionext*.

Lemma 1 (Termination)**.** *For a valid, acyclic Esterel program, getPrio() and getPrioNext() terminate. Furthermore, they do not generate a "Cycle detected!" error message.*

Proof. (Sketch) getPrio() produces an error (line 4) if it has not computed *n.prio* yet (checked in line 2) but has already been called (line 3) earlier in the call chain. This means that it has called itself via one of the calls to prioMax() or prioNextMax() (via getPrioNext()). An inspection of the calling pattern yields that an acyclic program in the sense of Definition 1 cannot yield a cycle in the recursive call chain. □

Lemma 2 (Fulfillment of Constraints)**.** *For a valid, acyclic Esterel program, the priority assignment algorithm computes an assignment that fulfills Constraints 1–4.*

Proof. (Sketch) First observe that—apart from the initialization in main()—each *n.prio* is assigned only once. Hence, when prioMax() returns the maximum of priorities for a given set of nodes, these priorities do not change any more. Therefore, the fulfillment of Constraint 1 can be deduced directly from getPrio. Similarly for Constraint 2. Analogously getPrioNext() ensures that Constraints 3 and 4 are met. □

Lemma 3 (Linearity)**.** *For a CKAG with N nodes and E edges, the computational complexity of the priority assignment algorithm is $\mathcal{O}(N + E)$.*

```
procedure genPrioCode()
  forall $n \in F$ do // Step 1
    forall $m \in n.suc_c$ do
      annotate corresponding PAR statement with $m.prio$

  forall $n \in N$ do // Step 2
    // Case p.prio $<$ n.prio impossible!
    $P := \{p \mid n \in p.suc_f, p.id = n.id\}$ // $id$ is the thread id
    $prio :=$ max($\{p.prio \mid p \in P\} \cup \{p.prionext \mid p \in P \cap D\}$)
    if $prio > n.prio$ then
      insert "PRIO $n.prio$" at $n$
      // If $n \in D$: insert before $n$ (eg, PAUSE)
      // If $n \in T$: insert after $n$ (eg, a label)

  forall $n \in D \cup F$ do // Step 3
    // Case $n.prio > n.prionext$ is already covered in Step 2
    if $n.prio < n.prionext$ then
      insert "PRIO $n.prionext$" before $n$
end
```

Figure 6. Algorithm to annotate code with priority settings according to CKAG node priorities.

Proof. (Sketch) Apart from the initialization phase, which has cost $\mathcal{O}(N)$, the cost of the algorithm is dominated by the recursive calls to getPrio(). The total number of calls is bounded by E. With an amortization argument, where the costs of each call are attributed to the callee, it is easy to see that the overall cost of the calls is $\mathcal{O}(E)$. □

Note also that while the size of the CKAG may be quadratic in the size of the corresponding Esterel program in the worst case, it is in practice (for a bounded abort nesting depth) linear in the size of the program, resulting in an algorithm complexity linear in the program size as well.

After priorities have been computed for each reachable node in the CKAG, we must generate code that ensures that each thread is executed with the computed priority. This task is relatively straightforward, Figure 6 shows the algorithm.

Another issue is the computation of thread *id*s, as these are also considered in scheduling decisions in case there are multiple threads of highest priority. This property is exploited by the scheduling scheme presented here, to avoid needless cycles. The compiler assigns increasing *id*s to threads during a depth-first traversal of the thread hierarchy; this is required in certain cases to ensure proper termination of concurrent threads [15].

6.3 Optimizations

The KEP3a compiler performs a dismantling, as a preprocessing step, to eliminate dependency cycles (see Section 3.3). After assigning priorities, the compiler tries again to "undismantle" compound statements whenever this is possible. This becomes apparent in the CKAG in Figure 1(c) for example in node L3; this AWAIT S is the undismantled equivalent of the lines 7–11 in Edwards02-dism.

The compiler suppresses PRIO statements for the main thread, because the main thread never runs concurrently to other threads. In the example, this avoids a PRIO 1 statement at label A3.

Furthermore, the compiler performs dead code elimination, also using the traversal results of the priority assignment algorithm. In the Edwards02 example, it determines that execution never reaches the infinite loop in lines 36–38 of Edwards02-dism, because the second parallel thread never terminates normally, and therefore does not generate code for it.

However, there is still the potential for further optimizations, in particular regarding the priority assignment. In the Edwards02 program, one could for example hoist the PRIO 2_{21} out of the enclosing loop, and avoid this PRIO statement altogether by just starting thread T2 with priority 2 and never changing it again. Even more effective would be to start T3 with priority 3, which would allow to undismantle L08–L12 into a single AWAIT.

7. Experimental Results

To validate the correctness of the KEP3a and its compiler and to evaluate its performance, we employed an evaluation platform whose structure is shown in Figure 7. The user interacts via a host work station with an FPGA Board, which contains the KEP3a as well as some testing infrastructure. First, an Esterel program is compiled into an KEP object file (.ko) which is uploaded to the FPGA board. Then, the host provides Input events to the KEP3a and reads out the generated Output events. This also yields the number of instructions per tick, from which we can deduce the worst case reaction time for the given trace. The input events can be either provided by the user interactively, or they can be supplied via a .esi file. The host can also compare the Output results to an execution trace (.eso). We use EsterelStudio V5.0 to compute trace files with state and transition coverage, except for the eight_but benchmark, for which the generation of the transition coverage trace took unacceptably long and we restricted ourselves to state coverage. This comparison to a reference implementation proved a very valuable aid in validating the correctness of both the KEP3a and its compiler.

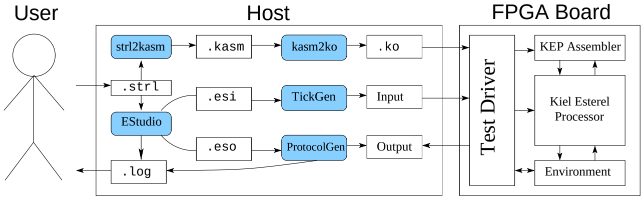


Figure 7. The structure of the KEP evaluation platform

For a comparison of the performance of the KEP3a and its compiler with another platform, we chose the MicroBlaze 32-bit soft COTS RISC processor core as the reference point. We use the CEC compiler 0.3, the Esterel Compiler V5.92, and the Esterel Compiler V7 to synthesize Esterel modules to C programs, which are then compiled onto the MicroBlaze via gcc version 2.95.3-4, using the default level 2 optimization.

Table 2 summarizes experimental results for a number of standard benchmarks [5, 2, 7]. To characterize each benchmark with respect to its use of concurrency and preemption constructs, the table lists the count and depth of them. For the KEP3a, the table lists the number of dependencies found, the used number of priority levels (the KEP3a provides up to 255), and the number of used PRIO instructions. We see that in most cases, the maximum priority used is three or less, indicating relatively few priority changes per tick. To assess the usefulness of providing different types of watchers, as has been described in Section 5.2, Table 2 also lists which how many watchers of each type were required. As it turns out, most of the preemptions could be handled by the cheapest Watcher type, the Thread Watcher. For example, in the case of the mca200 benchmark, this reduces the hardware requirements from 4033 slices (if all preemptions were handled by general purpose Watchers) to 3265 slices. Table 2 also compares the results for the completely dismantled ("unoptimized") and the partially undismantled ("optimized") version, as explained in Section 6.3. As the results indicate, the optimized version often uses significantly less Watchers than the unoptimized version. We also compare compilation times, from Esterel code to machine code, and notice that the KEP3a compiler is quite competitive with the synthesis onto the Microblaze.

Module Name	Instr's total abs.	CSs total abs.	CSs total ratio	CSs at same priority abs.	CSs at same priority rel.	PRIOs total abs.	PRIOs total rel.	CSs due to PRIO abs.	CSs due to PRIO rel.	CSs due to PRIO rel.
	[1]	[2]	[1]/[2]	[3]	[3]/[2]	[4]	[4]/[1]	[5]	[5]/[2]	[5]/[4]
abcd	16513	3787	4.36	1521	0.40	3082	0.19	1243	0.33	0.40
abcdef	29531	7246	4.08	3302	0.46	6043	0.20	2519	0.35	0.42
eight_but	39048	10073	3.88	5356	0.53	8292	0.21	3698	0.37	0.45
chan_prot	5119	1740	2.94	707	0.41	990	0.19	438	0.25	0.44
reactor_ctrl	151	48	3.15	29	**0.60**	0	0	0	0	-
runner	5052	704	**7.18**	307	0.44	0	0	0	0	-
example	208	60	3.47	2	**0.30**	26	0.13	9	0.15	0.35
ww_button	292	156	**1.87**	92	0.59	0	0	0	0	-
greycounter	160052	34560	4.63	14043	0.41	26507	0.17	12725	0.37	**0.48**
mca200	982417	256988	3.82	125055	0.49	242457	**0.25**	105258	**0.41**	0.43

Table 3. Analysis of context switches (CSs), in absolute numbers and relative. Minimal and maximal relative values are shown **bold**.

Module Name	Esterel LOC	MicroBlaze Code+Data (b)			KEP3a, unoptimized Code (w)		KEP3a, unoptimized Code+Data (b)		KEP3a, opt. Code (w)	
		V5	V7	CEC	abs.	rel.	abs.	rel.	abs.	rel.
	[1]	[2] (**best**)			[3]	[3]/[1]	[4]	[4]/[2]	[5]	[5]/[3]
abcd	160	**6680**	7928	7212	168	1.05	756	0.11	164	0.93
abcdef	236	9352	9624	**9220**	252	1.07	1134	0.12	244	0.94
eight_but	312	12016	**11276**	11948	336	1.08	1512	0.13	324	0.94
chan_prot	42	3808	6204	**3364**	66	1.57	297	0.09	62	0.94
reactor_ctrl	27	2668	5504	**2460**	38	1.41	171	0.07	34	0.89
runner	31	3140	5940	**2824**	39	1.22	175	0.06	27	0.69
example	20	2480	5196	**2344**	31	1.55	139	0.06	28	0.94
ww_button	76	6112	7384	**5980**	129	1.7	580	0.10	95	0.74
greycounter	143	**7612**	7936	8688	347	2.43	1567	0.21	343	1
mca200	3090	104536	77112	**52998**	8650	2.79	39717	0.75	8650	1

Table 4. Memory usage comparison between KEP and MicroBlaze implementations. "(b)" refers to measurements in bytes, "(w)" to words.

Table 3 analyzes the context switch (CS) activities in the benchmarks, for some specific test traces. For example, of the total of 292 instructions executed for the ww_button benchmark, there was a CS at about every other instruction, whereas for the runner benchmark, there was a CS roughly every seven instructions. This indicates that the fast, light-weight CS mechanism of the KEP3a is a key to performance for executing these types of reactive programs. Overall, between 30 and 60% of the CSs took place at the same priority, that is, because threads became inactive and another thread at the same priority took over. Some benchmarks did not require any PRIO instructions, for others they constituted up to 25% of the instructions executed. Up to 37% of CSs were due to PRIO instructions. Finally, for those benchmarks that included PRIO statements, less than half of the PRIO instructions actually resulted in a CS, indicating that a static schedule would have been comparatively inefficient.

Table 4 compares executable code size and RAM usage between the KEP3a and the MicroBlaze implementations. To assess the size of the KEP3a code relative to the Esterel source, we compare the code size in words to the Esterel Lines of Code (LOC, before dismantling, without comments), and notice that the KEP3a code is very compact, with a word count close to the Esterel source. For comparison with the Microblaze, we compare the size of Code + Data, in bytes, and notice that the KEP3a code is typically an order of magnitude smaller than the MicroBlaze code. Regarding the effectiveness of the compiler optimization, Table 4 indicates that this on average leads to an 10% memory reduction. Finally, the KEP3a implementation results on average in an 83% reduction of memory usage (codes and RAM size) when compared with the best result of the MicroBlaze implementation.

	Esterel					KEP3a (unoptimized|optimized)								MicroBlaze		
Module	Threads			Preemptions		CKAG				Preemption handled by			Compiling	Compiling		
Name	Count	Max	Max	Count	Max	Nodes	Dep.	Max	PRIO		Local	Thread	time (sec)	time (sec)		
		depth	conc.		depth		count	priority	instr's	Watcher	Watcher	Watcher		V5	V7	CEC
abcd	4	2	4	20	2	211	36	3	30	0	4|3	16|11	0.15	0.12	0.09	0.30
abcdef	6	2	6	30	2	313	90	3	48	0	6|5	24|17	0.21	0.71	0.46	0.96
eight_but	8	2	8	40	2	415	168	3	66	0	8|7	32|23	0.26	0.99	0.54	1.25
chan_prot	5	3	4	6	1	80	4	2	10	0	0	6|4	0.07	0.35	0.35	0.43
reactor_ctrl	3	2	3	5	1	51	5	1	0	0	1|0	4	0.06	0.29	0.31	0.36
runner	2	2	2	9	3	61	0	1	0	3|2	1	5|3	0.05	0.30	0.34	0.40
example	2	2	2	4	2	36	2	3	6	0	1	3|2	0.05	0.28	0.31	0.31
ww_button	13	3	4	27	2	194	0	1	0	0	5	22|10	0.10	0.44	0.40	0.64
greycounter	17	3	13	19	2	414	53	6	58	0	4	15	0.34	0.57	0.43	0.75
mca200	59	5	49	64	4	11219	129	11	190	2	14	48	11.25	69.81	12.99	7.37

Table 2. Experimental results of the KEP and its Esterel compiler.

	MicroBlaze						KEP3a, unoptimized				KEP3a, optimized			
Module	WCRT			ACRT			WCRT		ACRT		WCRT		ACRT	
Name	V5	V7	CEC	V5	V7	CEC	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
		[1] (**best**)			[2] (**best**)		[3]	[3]/[1]	[4]	[4]/[2]	[5]	[5]/[3]	[6]	[6]/[4]
abcd	1559	**954**	1476	1464	**828**	1057	135	0.14	87	0.11	135	1	84	0.97
abcdef	2281	**1462**	1714	2155	**1297**	1491	201	0.14	120	0.09	201	1	117	0.98
eight_but	3001	**1953**	2259	2833	**1730**	1931	267	0.14	159	0.09	267	1	153	0.96
chan_prot	754	**375**	623	683	**324**	435	117	0.31	60	0.19	117	1	54	0.90
reactor_ctrl	487	**230**	397	456	**214**	266	54	0.23	45	0.21	51	0.94	39	0.87
runner	566	**289**	657	512	**277**	419	36	0.12	15	0.05	30	0.83	6	0.40
example	467	**169**	439	404	**153**	228	42	0.25	24	0.16	42	1	24	1
ww_button	1185	**578**	979	1148	**570**	798	72	0.12	51	0.09	48	0.67	36	0.71
greycounter	1965	**1013**	2376	1851	**928**	1736	528	0.52	375	0.40	528	1	375	1
mca200	75488	29078	**12497**	73824	24056	**11479**	2862	0.23	1107	0.10	2862	1	1107	1

Table 5. The worst-/average-case reaction times (in clock cycles) for the KEP3a and MicroBlaze implementations, in absolute and relative values.

	MicroBlaze	KEP3a		Ratio	
Module	(82mW@50MHz)	(mW)		(KEP to MB)	
Name	Blank	Peak	Blank	Peak	Blank
abcd	69	13	8	0.16	0.12
abcdef	74	13	7	0.16	0.09
eight_but	74	13	7	0.16	0.09
chan_prot	70	28	12	0.34	0.17
reactor_ctrl	76	20	13	0.24	0.17
runner	78	14	2	0.17	0.03
example	77	25	9	0.30	0.12
ww_button	81	13	4	0.16	0.05
greycounter	78	44	33	0.54	0.42

Table 6. The energy consumption comparison between KEP and MicroBlaze implementations.

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 7. Extending a KEP3a to different threads.

The improvement in execution time of the KEP3a implementation is shown in Table 5. Comparing with the best result of the MicroBlaze implementations, the KEP3a typically obtains more than 4x speedup for the WCRT, and more than 5x for the Average Case Reation Time (ACRT).

To compare the energy consumptions, we choose the Xilinx 3S200-4ft256 as FPGA platform. This requires an additional 37mW as quiescent power for the chip itself. Based on the findings presented in Table 5, we calculate the clock frequencies needed for the KEP3a and MicroBlaze systems to achieve the same WCRT, and then estimate their energy consumptions by Xilinx WebPower Version 8.1.01. Table 6 shows that the KEP3a reduces energy usage on average by 75%. The reduction becomes even more significant if all environment inputs are absent, a rather frequent case; in this case, the KEP3a achieves 86% power savings.

The KEP3a is highly configurable, including the possible degree of concurrency. To assess the resource efficacy of the multi-threaded approach relative to multi-processing approach, we generated KEP3a versions with a maximal thread number varying between 2 and 120. All versions are configured with 2 Watchers, 8 Local Watchers, and 48 valued I/O signals. The clock rate does not vary significantly, it is around 60 MHz; one instruction takes three clock cycles. Table 7 shows the corresponding resources usages. The hardware usage increases only 4x when the concurrency increases 60x when measured in slices, and even just 1.4x when measured in equivalent gates. The implementation is based on the Xilinx 3S1500-4fg676 FPGA. For the comparison, the MicroBlaze which with the same memory size (BRAM) employs 309k gates.

8. Conclusions & Outlook

We have presented the KEP3a, a multi-threaded processor, which allows the efficient execution of concurrent Esterel programs. It provides significant improvements over earlier reactive processing approaches, mainly in terms of completeness of the instruction set and its efficient mapping to hardware. The multi-threaded approach poses specific compilation challenges, in particular in terms of scheduling, and we have presented an analysis of the task at hand as well as an implemented solution. As the experimental comparison with a 32-bit commercial RISC processor indicates, the approach presented here has advantages in terms of memory use, execution speed, and energy consumption.

However, there is still room for optimization. The thread scheduling problem is related to the problem of generating statically scheduled code for sequential processors, for which Edwards has shown that finding efficient schedules is NP hard [10]. We encounter the same complexity, but our performance metrics is a little different. The classical scheduling problem tries to minimize the number of context switches. On the KEP3a, context switches are free, because no state variables must be stored and resumed. However, to ensure that a program meets its dependency-implied scheduling constraints, threads must manage their priorities accordingly, and it is this priority switching which contributes to code size and costs an extra instruction at run time. Minimizing priority switches is related to classical constraint-based optimization problems as well as to compiler optimization problems such as loop invariant code motion.

This paper has made the case for a custom processor design for the efficient execution of concurrent reactive programs. However, the underlying model of computation, with threads keeping their individual program counters and a priority based scheduling, could also be emulated by classical processors. This would be less efficient than a custom processor, but still could take advantage of the compact program representation developed here. The CKAG could serve as a basis for traditional intermediate language (C) code generation; the main issue here would be an efficient implementation of the scheduler. However, as one would not have to run the scheduler at every instruction, as currently done by the KEP3a, but only at fork or delay nodes, this might still be reasonably efficient. Furthermore, it would be interesting to implement a virtual machine that has an instruction set similar to the KEP3a; see also the recent proposal by Plummer *et al.* [20]. Finally, it should be a very interesting project to implement the KEP3a itself in Esterel, which would not only make an interesting benchmark, but could also be used to synthesize a virtual machine.

We are also investigating to augment the KEP3a with external hardware to speed up the computation of signal expressions [14]. Another interesting problem raised by the KEP3a's multi-threaded architecture is the analysis of its WCRT. Finally, we are also interested in developing an advanced energy management methodology, which is based on the WCRT information, to save the power consumption further.

Acknowledgments

This work has benefited from discussions with many people, in particular Michael Mendler and Stephen Edwards. We would also like to thank Mary Hall, Jan Lukoschus, Jan Täubrich, Claus Traulsen, and the reviewers for providing valuable feedback on this manuscript.

References

[1] C. André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[2] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. M. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, Apr. 1997.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Jan. 2003.

[4] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.

[5] G. Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[6] G. Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[7] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[8] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In F. Maraninchi, A. Girault, and E. Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, July 2002.

[9] S. A. Edwards. CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[10] S. A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), Feb. 2002.

[11] S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.

[12] S. A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'04)*, Barcelona, Spain, Mar. 2004.

[13] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

[14] S. Gädtke, X. Li, M. Boldt, and R. von Hanxleden. HW/SW Co-Design for a Reactive Processor. In *Proceedings of the Student Poster Session at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[15] X. Li, M. Boldt, and R. v. Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technischer Bericht 0603, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, May 2006. `http://www.informatik.uni-kiel.de/reports/2006/0603.html`.

[16] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. v. Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, Sept. 2005. ACM Press.

[17] X. Li and R. von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[18] J. Lukoschus. *Removing Cycles in Esterel Programs*. PhD thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, July 2006.

[19] M. D. Natale, A. Sangiovanni-Vincentelli, and F. Balarin. Task scheduling with RT constraints. In *DAC '00: Proceedings of the 37th Conference on Design Automation*, pages 483–488, New York, NY, USA, 2000. ACM Press.

[20] B. Plummer, M. Khajanchi, and S. A. Edwards. An Esterel virtual machine forembedded systems. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, Mar. 2006.

[21] D. Potop-Butucaru and R. de Simone. *Optimization for faster execution of Esterel programs*, pages 285–315. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[22] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, Pisa, Italy, Sept. 2004.

[23] L. H. Yoong, 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, Mar. 2006.