Multi-Threaded Reactive Programming— The Kiel Esterel Processor

Xin Li and Reinhard von Hanxleden, *Member, IEEE*

Abstract—The Kiel Esterel Processor (KEP) is a multi-threaded reactive processor designed for the execution of programs written in the synchronous language Esterel. Design goals were timing predictability, minimal resource usage, and compliance to full Esterel V5. The KEP directly supports Esterel's reactive control flow operators, notably concurrency and various types of preemption, through dedicated control units. Esterel allows arbitrary combinations and nestings of these operators, which poses particular implementation challenges that are addressed here. Other notable features of the KEP are a refined instruction set architecture, which allows to trade off generality against resource usage, and a Tick Manager that minimizes reaction time jitter and can detect timing overruns.

Index Terms—Reactive systems, concurrency, multi-threading, synchronous languages, Esterel, low-power design, predictability

1 Introduction

Many embedded systems belong to the class of *reactive systems*, which continuously react to inputs from the environment by generating corresponding outputs. The programming of reactive systems typically requires the use of non-standard control flow constructs, such as concurrency and exception handling. Most programming languages, including languages such as C and Java that are commonly used in embedded systems, either do not support these constructs at all, or their use induces non-deterministic program behavior, regarding both functionality and timing.

To address this difficulty, the synchronous language Esterel [2] has been developed to express reactive control flow in a concise, deterministic manner. This is valuable for the designer, but also poses implementation challenges. As Esterel is a domain-independent programming (specification) language, there are a number of implementation alternatives, each with its advantages and drawbacks, see also Table 1. An Esterel program is typically validated via a simulation-based tool set, and then synthesized to an *intermediate language*, *e.g.*, C or VHDL [1]. 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 [5].

- *R. von Hanxleden is with the Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany. E-Mail: rvh@informatik.uni-kiel.de.*
- *X. Li is with the Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities, Minneapolis, Minnesota, USA. E-Mail: lixxx914@umn.edu.*

Manuscript received June 8, 2007; revised Oct. 30, 2009; accepted Nov. 4, 2009; published online XXX.
Recommended for acceptance by XXX
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-06-0223. Digital Object Identifier no. XXX. .

Reactive programs are often characterized by very frequent context switches; as it turns out, a context switch after every three or four instructions is not uncommon [8]. 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 may be affected by a preemption.

To address these difficulties, the recently emerging *reactive processing* approach strives for a direct implementation of Esterel's control flow and signal handling constructs. This provides hardware support for handling reactive control flow, alleviating the need for a compiler that sequentializes the code or for an OS that emulates concurrency. In this paper, we present the Kiel Esterel Processor (KEP) reactive architecture. The development of the KEP was driven by the desire to achieve predictable, 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 KEP, which allows the mapping of Esterel programs into compact machine code while keeping the processor compact. To keep the KEP simple and light-weight, it currently does not employ classical acceleration mechanism such as pipelining, other forms of instruction level parallelism, or caching. Such mechanism can be combined with reactive processing [9], but typically there is a trade-off between average-case performance and predictability. Still, the worst case reaction time of the KEP is typically improved by 4x compared to the MicroBlaze, a COTS RISC processor core, and energy consumption is also typically reduced to a quarter; see also Sec. 6.4.

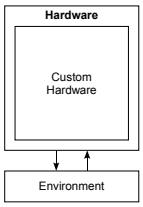
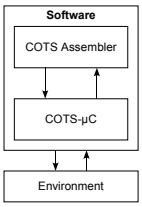
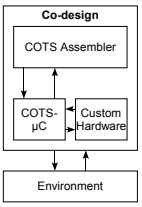
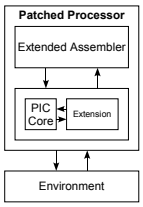
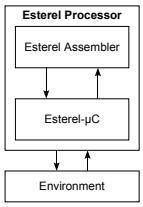
		Hardware	Software	Co-design	Patched Processor	Custom Processor
Architecture						
Speed		++	−	+	+	+
Selected References		Berry [1], Edwards [3]	Berry *et al.* [2], Edwards *et al.* [4]	Balarin *et al.* [5]	Roop *et al.* [6]	Li *et al.* [7]
Flexibility		− −	++	−	+/−	+
Esterel Compliance		++	++	+/−	−	++
Cost	Logic Area	++/−	+	+	− −	+/−
	Memory	++	− −	−	+	+
	Power Usage	++	−	−	− −	+
Appl. Design Cycle		− −	++	+/−	++	++

TABLE 1
Comparison of Esterel implementation alternatives. ++ represents best; − − means worst.

This paper presents a comprehensive overview of the KEP architecture and how it meets the challenge to accurately and efficiently implement the rich, strictly synchronous semantics of the Esterel language. Beyond earlier publications [7], [8], [10] that covered different aspects of the KEP as realized in earlier generations (see Sec. 3), this paper also presents a detailed treatment of the interaction of concurrency and preemption (Sec. 5/5.2), and a fairly extensive review of the development of reactive processing so far. A full presentation of the concrete design of the KEP, its validation and the experimental evaluation, in more detail than is possible here, can be found in the dissertation of the first author [11]. Note also that this paper focuses on the architecture and the ISA of the KEP. Closely related are the issues of code generation for the KEP and Worst Case Reaction Time (WCRT) analysis, both of which are covered elsewhere in detail [12], [13].

The rest of this paper is organized as follows. The next section provides some basics on the Esterel language. Sec. 3 discusses related work. Sections 4 and 5 present the instruction set and the architecture of the KEP. The validation platform and experimental results are presented in Sec. 6. The paper concludes in Sec. 7.

2 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). The test for the presence of a signal is per default *non-immediate*; for example, an await S first pauses for one tick, and only from the next tick on tests for the presence of S. There are also *immediate* variants, for example, await immediate S tests for S from the same instant onwards that the statement is entered. 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.

2.1 Reactive control flow

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, abortion and suspension. 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 tick when the trigger event occurs.

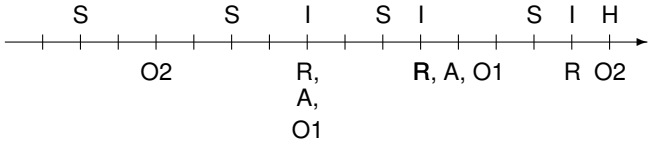
Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are specific rules 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 will be weakly aborted; if concurrent threads

```
1  module EXAMPLE:
2  input S, I, H;
3  output O1, O2;
4  signal A,R in
5  % −−−−−−−− Thread 0
6  every S do
7    trap T1 in
8      trap T2 in
9  % −−−−−−−− Thread 1
10       [ await I;
11         weak abort
12           suspend
13             sustain R;
14           when H;
15         when immediate A;
16         emit O1;
17         exit T1;
18       ||
19 % −−−−−−−− Thread 2
20         await 2 tick;
21         present R then
22           emit A;
23         end present;
24         exit T2; ];
25 % −−−−−−−− Thread 0
26       end trap;
27       emit O2;
28     end trap;
29 end every;
30 end signal;
31 end module
```

(a) Esterel program

```
1            INPUT S, I, H
2            OUTPUT O1, O2
3            SIGNAL A, R
4  % −−−−−−−−−− Thread 0
5            EMIT _TICKLEN, #20
6            AWAIT S
7  A0:       ABORT S, A1
8  T1S: T2S: PAR 3, P1
9            PAR 2, P2
10           PARE P3
11 % −−−−−−−−−− Thread 1
12 P1:       AWAIT I
13           WABORTI A, A2
14           SUSPEND H, A4
15 A3:       EMIT R
16           PRIO 1
17           PRIO 3
18           PAUSE
19           GOTO A3
20 A4: A2:   EMIT O1
21           EXIT T1E,T1S
22 % −−−−−−−−−− Thread 2
23 P2:       LOAD _COUNT, #2
24           AWAIT TICK
25           PRESENT R, A5
26           EMIT A
27 A5:       EXIT T2E,T2S
28 % −−−−−−−−−− Thread 0
29 P3:       JOIN 0
30 T2E:      EMIT O2
31 T1E:      HALT
32 A1:       GOTO A0
```

(b) KEP assembler



(c) Sample execution trace, with inputs above and outputs below logical tick time line

Fig. 1. EXAMPLE: an Esterel module illustrating Esterel parallel, preemption, and exception statements.

execute multiple exit instructions in the same tick, the outermost trap will take priority.

2.2 An Example

Let us consider the EXAMPLE module in Fig. 1a. After the input/output/local signal declarations, an every block restarts its body whenever the input signal S is present—except for the initial tick, when S is ignored, as the every is not "immediate." Inside the every block are two nested trap scopes, an outer trap triggered by T1 (via an exit T1 exception) and an inner T2 trap. The body of the inner trap contains two parallel threads. Thread 1 initially waits for the input signal I. Once I has become present (non-immediately), the sustain R_{13}[1] statement continuously emits the local signal R. However, that sustain is weakly aborted by A (immediately), and suspended by H (non-immediately). Once A has triggered the abortion, O1 is emitted and the exception T1 is thrown. Thread 2 initially idles for two ticks, then emits A if R is present, and exits the T2 trap.

A possible execution trace is shown in Fig. 1c. All signals are absent at the initial tick. At the second tick, the presence of input signal S triggers the start of the every body, which spawns Threads 1 and 2. Thread 1 stays at the await I, since this is non-immediate, and Thread 2 stays at the await 2 tick. At the third tick, Thread 1 again cannot proceed, since I is absent, and Thread 2 stays the second tick at the await 2 tick. At the fourth tick, Thread 1 again does not get an input I; Thread 2 can proceed, detects R as absent, throws exception T2, and terminates. This exception aborts Thread 1 and transfers control to the end of the trap T2 scope, hence O2 gets emitted and control moves back to the every S loop. The other possible behaviors that follow the next occurrences of S are shown in the remainder of the time line. For a detailed understanding, the reader might also consult the Esterel primer [14].

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

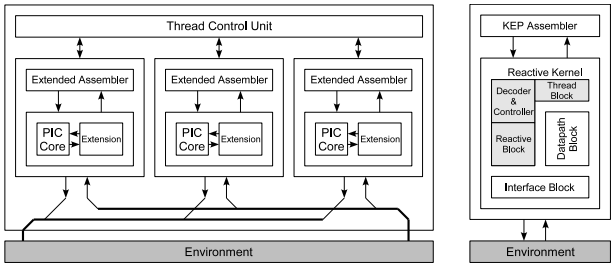
3 Related Work

There is a by now extensive body of research on the efficient compilation of Esterel in the various domains, a full discussion of which is beyond the scope of this paper. See for example Potop-Butucaru/Edwards [15] for a good overview of software synthesis approaches. We here focus on existing work in the field of reactive processing, which is a rather recent development. An earlier overview was presented in [16].

3.1 Sequential reactive processing

The first reactive processor, called REFLIX [6], was presented by Salcic, Roop *et al.* in 2002. The REFLIX is a *patched processor*, combining a traditional soft micro controller core (FLIX) with a custom hardware block that extends the instruction set of the FLIX by certain new, Esterel-like instructions; see also Table 1. Although the Esterel-style statements (instructions) supported by the REFLIX were very limited, it performed better than its competitors, *i.e.*, the FLIX and other micro controllers. The REPIC [17] replaced the FLIX by the PIC processor, which is popular in the industry control domain. Both of these patched processors are limited by the control path of the traditional processor. This prevents for example the proper handling of nested traps, as the control path there depends on address ranges and parallel relations of the traps.

In 2004, the authors presented the first prototype of the KEP [18], now referred to as the "KEP1," to our knowledge the first *custom reactive processor* fully designed from scratch. The KEP1 was also the first to correctly handle weak and strong abortion. It included Watcher units that handle such abortions concurrently to



(a) Multi-processing (b) Multi-threading

Fig. 2. Comparison of concurrent reactive architectures.

the regular control flow, *i.e.*, without the need to execute extra instructions to check for the triggering of abortions (see also Sec. 5.2. However, the KEP1 did not provide full concurrency, and logic and arithmetic expression were also not supported.

In 2005, Z. Salcic *et al.* presented the REMIC, another custom processor [19]. In the same year, the KEP2 improved over the KEP1 in that it includes an interface block that supports the PRE-operator, and can handle further Esterel-constructs such as variables and local signals [10]. Furthermore, it contains an ALU and supports some classical logic and arithmetic expression. The KEP2 also includes a Tick Manager, which can provide a constant logical tick length and detects timing overruns; see also Sec. 5.5.

3.2 Concurrent reactive processing

Perhaps the most distinguishing feature for reactive processors is whether and how they handle concurrency. The first generations of reactive processors did not support Esterel's concurrency operator directly. Executing concurrent Esterel programs thus required to transform them into an equivalent program with a flattened state space, *i.e.*, was sequentialized by constructing a "product automaton."

The EMPEROR [20], [21] introduced the *multi-processing* approach for handling concurrency directly. Here, every Esterel thread is mapped onto an independent processor to be executed, and a thread control unit handles the synchronization and communication between processors; see also Fig. 2a. The EMPEROR uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential to speed up execution relative to single-processor implementations. However, this execution model potentially requires to replicate parts of the control logic at each processor, and is thus relatively hardware-intensive. Furthermore, it is difficult to support the arbitrary nesting of concurrency and preemption. A more efficient concurrency implementation approach for reactive processing appears to be *multi-threading*, which was first employed by the KEP3, presented in 2006 [7]; see also Fig. 2b and further explanations in Sec. 5.1. As illustrated in Sec. 6, this approach scales well to high degrees of concurrency with minimal resource overhead. By now the multi-threading approach has also been adopted by the STARPro [9], which further improves performance by pipelining.

Later in 2006, the KEP3a and its compiler were presented [8]. The KEP3a has improved over the KEP3 in that it supports exception handling and provides context-dependent preemption handling instructions. This paper presents version 4 of the KEP (KEP4), which compared to earlier generations has an enriched control path for handling advanced mixed Esterel control structures, and supports numerous options for generating processor configurations. The KEP4 and the strl2kasm compiler fully support Esterel V5. The subsequently evolved Esterel V7 has numerous extensions, mainly to support hardware design, but has the same underlying reactive control flow operations; hence, extending the KEP approach for Esterel V7 appears to be mostly a compiler issue and should not require fundamental changes of the architecture described here.

3.3 Compilation, WCRT analysis, co-design

Since multi-processing and multi-threaded reactive processors employ different strategies to handle Esterel concurrency, their compilers, which synthesize an Esterel program to the target reactive processor codes, also use different approaches to implement the communication between threads.

For multi-processing, the EMPEROR Esterel Compiler 2 (EEC2) [21] is based on a variant of the *graph code* (GRC) format [15], 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 to take compile-time scheduling knowledge into account, but replaces it by repeating cycles through all threads until all signal values have been determined. Hence the compiler needs to generate sync instructions, to ensure that signals are not tested before they are emitted [20]. In comparison, the multi-threaded implementation approach implements interleaving by inserting priority setting instructions at the context switch point.

The compiler for the multi-threaded KEP employs a priority assignment approach that makes use of a novel concurrent control flow graph, the Concurrent KEP Assembler Graph (CKAG). The worst-case size of the CKAG is quadratic in the size of the corresponding Esterel program; in practice, namely for a bounded abort nesting depth, it is linear [12]. 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. The compilation for the KEP is further summarized by Li *et al.* [8] and presented in detail by Boldt [12].

Since one key characteristic of the KEP is its timing predictability, it is feasible to perform a conservative, yet fairly accurate analysis of its WCRT. A first WCRT technique was presented in 2005 [10], for the (sequential) KEP2. An extension to concurrent WCRT analysis was later presented by Boldt *et al.* [13]. The analysis of the WCRT is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. The analysis is performed on the CKAG. In a first step one computes whether concurrent threads terminate instantaneously, thereafter one computes for each statement how many instruction are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program.

The KEP has also been employed as a platform for HW/SW co-design. Gädtke *et al.* [22] present an approach to accelerate reactive processing via an external logic block that handles complex signal expressions. An Esterel program is synthesized into a software component, running on the KEP, and a hardware component, consisting of simple combinational logic. The transformation process involves a two-step procedure, which first partitions the program at the source level and subsequently performs the synthesis. An intermediate logic minimization, at the source code level, facilitates the synthesis of compact logic blocks.

3.4 Further related work

The KEP series of processors focuses on reactive control flow. There are also other architectures, such as the Kiel Lustre Processor (KLP) [23], that focus on data-flow and its efficient execution via parallelization.

Timing predictability is also the focus of the Precision Timed Architecture [24] developed at UC Berkeley and Columbia University. This architecture is also multi-threaded, with true parallelism, and employs physical time for thread synchronization. The PRET-C proposal [25] extends a traditional core with an external thread scheduling unit, and uses a synchronous programming model.

Related to the KEP ISA, there also have been recent proposals for expressing synchronous concurrency by a small set of operators embedded in a host language, such as LuSteraL [26] or SyncCharts in C [27]; see also the Conclusions.

4 The KEP Instruction Set Architecture

At the Esterel level, one distinguishes *kernel statements* and *derived statements*; the derived statements are basically syntactic sugar, built up from the kernel statements. Any set of Esterel statements from which the remaining statements can be constructed could be considered a valid set of kernel statements. The accepted set of Esterel kernel statements has indeed 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 kernel statements from the v5 standard [14]. The process of expanding derived statements into equivalent, more primitive statements—which may or may not be kernel statements—is also called *dismantling*. 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 KEP ISA is summarized in Table 2, which also illustrates the relationship between Esterel statements and the KEP instructions. The KEP uses a 36-bit wide instruction word and a 32-bit data bus. The corresponding instruction encoding is described elsewhere [11]. The KEP ISA has the following characteristics:

- All the kernel Esterel statements, and some frequently used derived statements, can be mapped to KEP instructions directly. For the remaining Esterel (V5) statements there exist dismantling rules that allow the compiler to generate KEP code, including general signal expressions (see [12]).
- The control statements are fully orthogonal, their behavior matches the Esterel semantics in all execution contexts.
- Common Esterel expressions, in particular all of the delay expressions (*i.e.*, standard, immediate, and count delays), can be represented directly. 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 KEP also handles schizophrenic programs correctly—if an Esterel statement must be executed multiple times within a tick, the KEP simply does so [8].

4.1 General Code Generation

An Esterel program is compiled into a KEP assembler program (.kasm) by the KEP Esterel compiler (strl2kasm) [12], which uses the front-end of the CEC for parsing and module expansion. In a second step, the KEP assembler compiler (kasm2ko) [28] compiles the assembler program into binary machine code. This includes, for example, the mapping of input/output/local signals to signal registers, and the selection of appropriate Watchers (see Sec. 5.2). The compiler also detects

Mnemonic, Operands	Esterel Syntax	Notes
INPUT[V] S	input S [:integer]	Input declaration.
OUTPUT[V] S	output S [:integer]	Output declaration.
SETV S, #*data*|*reg*		Set the initial value of S; similar to EMIT, but does not affect presence.
PAR *Prio*, *startAddr* [, *ID*] PARE *endAddr* JOIN *Prio*	[p_1 || … || p_n]	Fork and join, specifying the address range and the priority for each forked thread; see also Sec. 5.1. Optionally, one can specify the *ID* of the created thread.
PRIO *Prio*		Set the priority of the current thread.
[L|T][W]ABORT [*n*,] *Sexp*, *endAddr*	[weak] abort … when n *Sexp*	If *Sexp* is present, strongly/weakly abort the block ranging up to *endAddr*. The prefix [L|T] denotes the type of watcher to use, see also Sec. 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*	If *Sexp* is present, suspend the block ranging up to *endAddr*.
EXIT *endAddr*, *startAddr*	**trap** *T* **in** … **exit** *T* … **end trap**	Exit from a trap of specified 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. A compound statement, bracketed by CAWAITS and CAWAITE, with one CAWAIT[I] instruction per signal.
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*.
EMIT _TICKLEN, #*data*|*reg*		Set the tick length.
LOAD _COUNT, n		Load data for count delays.
LOAD _UINT32REG, #*data32*		Load a 32-bit immediate data to an intermediate register.
CLRC/SETC		Clear/set carry bit.
LOAD *reg*, *n*	*reg* := *n*	Load/store register.
{SR[C]|SL[C]|NOTR} *reg*		Shift (with carry)/negate.
{ADD[C]|SUB[C]|MUL} *reg*, *n*	+, -, *	Add, subtract (with carry), multiply. Division must be emulated.
{ANDR|ORR|NOTR|XORR} *reg*, *n*	and, or, not, xor	Logical operations.
CMP[S] *reg*, *n* JW *cond*, *addr*	>, <, <=, >=, <>, =	Compare (with sign), branch conditionally.

TABLE 2
Overview of the KEP 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.

whether the targeted KEP version does not have enough resources available, *e.g.*, not enough watchers or signals.

The KEP assembler code for the EXAMPLE is shown in Fig. 1b. Similar to the Esterel module, a KEP program always starts at the input/output definition. Lines 1 to 3 define input signals S and I, output signal O, and local signals A and R. The following EMIT _TICKLEN, #20 instruction assigns the tick length of this program as fixed 20 instruction cycles, as determined by the WCRT analysis [13]; see also Sec. 5.5.

For the program body, the generation of the KEP assembler for the EXAMPLE module is in general straightforward. As mentioned before, most common Esterel statements can almost literally be translated into corresponding KEP instructions, and there are direct equivalence rules for the remaining statements. In EXAMPLE, two dismantling rules, *i.e.*, the every translation rule and the sustain translation rule are employed. Note that due to the signal dependencies involving R, we cannot use the KEP's atomic SUSTAIN instruction [12].

4.2 Concurrency

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into KEP assembler as follows. First, threads are *forked* by a sequence of n PAR instructions and one PARE instruction. Each PAR

instruction creates one thread and assigns it a non-negative priority *Prio* and a start address *startAddr*. 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 as *endAddr* in a PARE instruction. A *thread address range* ranges from the start address to the end address. 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 Fig. 1b illustrates this: lines 12–21 constitute Thread 1, Thread 2 spans lines 23–27, and the remaining instructions belong to the main thread, Thread 0.

The main thread always has priority 1, which is the lowest possible priority. The priority of the other threads 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 thread 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 priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. However, 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 Columbia Esterel Compiler (CEC) [4]; see Li *et al.* [8] for further elaborations on causality/constructiveness and static vs. dynamic scheduling, and Lukoschus *et al.* [29] for an approach to expand cyclic, yet constructive programs into equivalent acyclic programs.

4.3 Handling Signal Dependencies

The signal-based communication employed by Esterel demands that signals have a unique presence/absence status throughout a tick, which is also why this strictly synchronous semantics is sometimes referred to as fixed-point semantics. This implies that a signal status should only be tested/read (*e.g.*, with PRESENT) once all potential emitters/writers (*e.g.*, EMIT) have executed. Assuring this also allows a proper reaction to signal absence. We also say that there is a *dependency* between the statements that (may) emit some signal (the *dependency sources*) and the statements where that signal is tested (the *dependency sinks*) [30].

In a concurrent setting, this implies that threads must be executed in an order that obeys these dependencies. Hence, a non-trivial aspect of code generation is the assignment of thread ids and priorities, as these govern the thread scheduling (see also Sec. 5.1). To understand how these priorities are assigned, we consider the thread scheduling constraints that must be obeyed to run the EXAMPLE module faithful to Esterel's semantics. The two threads enclosed in the every block can communicate back and forth via the R and A signals, within a logical tick, which makes thread scheduling non-trivial.

First, let us consider the dependency involving R. It is clear which instruction is the dependency source: the EMIT R_{15} instruction. It is also obvious that the PRESENT R_{25} instruction is the dependency sink. This allows us to formulate the first dependency present in the EXAMPLE module: whenever the EMIT R_{15} and the PRESENT R_{25} instructions are executed within the same logical tick, the execution of the EMIT must precede the execution of the PRESENT.

As for the dependency involving A, its dependency source is the EMIT A_{26} in Thread 2. However, it is less obvious which is the dependency sink, which we have defined above as the "statements where these signals are tested." Thread 1 reacts to A when it has entered the weak abort block, in that case A triggers the abort. Hence, whenever we execute a statement in that block ranging from WABORTI_{13} to the label $A2_{20}$, we should also watch for the presence of A. However, closer inspection yields that as this is a weak abort, it suffices to check at the end of each logical tick whether the block is aborted, that is, whenever we execute a delayed instruction. In this case, the only delayed instruction in the abort block is the PAUSE$_{18}$, which therefore constitutes the dependency sink for A.

In the EXAMPLE, the first dependency is met by starting Thread 1 with a higher priority (3) than Thread 2 (priority 2). The second dependency is met by the PRIO 1_{16} and PRIO 3_{17} instructions, which hand control from Thread 1 to Thread 2 and back.

Dependency analysis at the Esterel level is decidable (unlike at the KASM level), but a proper analysis that covers all aspects of the Esterel language is rather intricate. A detailed treatment of this is found in the documentation of the strl2kasm compiler [12].

5 The KEP Processor Architecture

Naturally, a given application has specific requirements regarding the computational resources. In the KEP, these resources include for example thread management and preemption capabilities. The KEP design is freely configurable, and scalable to arbitrary degrees of concurrency, preemption nestings, signal counts, etc. A KEP can thus be configured specifically for a given application. However, just as one may use a classical processor with some fixed, conservatively large memory resources for a range of applications, one may also use a "typical" KEP configuration without detailed prior

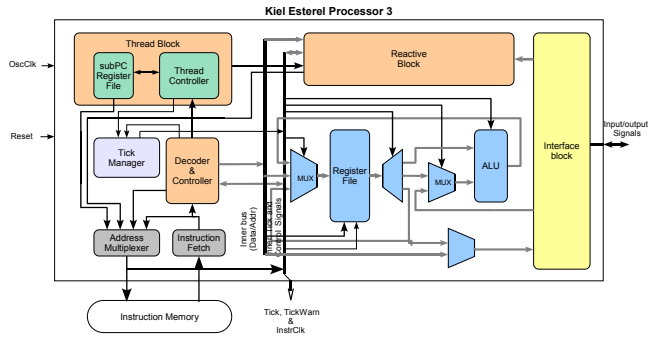
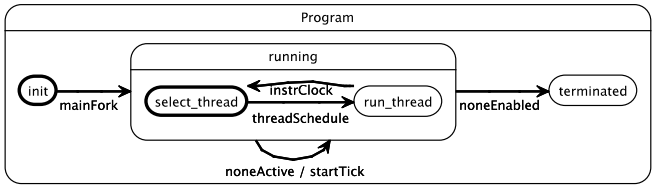
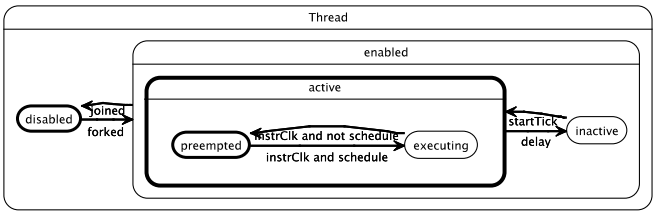


Fig. 3. Overview of the KEP architecture.



(a) Status of the whole program, as managed by the Thread Block



(b) Execution status of a single thread

Fig. 4. Execution model of the KEP.

knowledge about the application. For example, all the benchmarks used here have been done with a fixed configuration, except when assessing hardware scalability.

The architecture of the KEP, shown in Fig. 3, is inspired by the three layers that constitute a reactive program [2], *i. e.*, the *interface* layer, the *reactive kernel*, and the *data handling* layer. An Interface Block handles input reception and output production. The classical computations are performed by the Data Handling Block, consisting of the Register File, ALU and related components. The implementation of Esterel's reactive control statements relies on the cooperation of the KEP's Decoder & Controller, Reactive Block and Thread Block, which together form the Reactive Core (RC). The RC 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 KEP employs a multi-threaded architecture. Threads are scheduled in an interleaved fashion according to their statuses and dynamically changing priorities. The context of each thread is very light weight, it mainly consists of a program counter (PC), its priority, and two status flags (see below). All data are shared, consistent with Esterel's broadcast semantics. The scheduler is very light-weight. 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 (see Sec. 4.3), 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.

The Thread Block is responsible for managing threads, as illustrated in Fig. 4a. The SyncChart formalism [31] used here[2] consists of hierarchical finite state machines, bold state borders denote initial state at each hierarchy level. Upon program start, the main thread is enabled (forked), and the program is considered *running*. Subsequently, for each instruction cycle (instrClock), the Thread Block decides which thread ought to be scheduled for execution in this instruction cycle. It schedules the thread with the highest priority among all active threads; if there are multiple threads that have highest priority, the thread with the highest id is scheduled. If there are still enabled threads, but none is active anymore (see below), the next tick is started. If no threads are enabled anymore, the whole program is *terminated*.

The execution status of a single thread is illustrated in Fig. 4b. 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 (Thread 0) is enabled. Other threads become enabled whenever they are forked, and they become disabled again when they are joined after finishing all statements in their body, or when the preemption control tries to set its program 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*). Active threads are initially *preempted*, and become *executing* when they are scheduled.

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 allows a simple solution for handling arbitrary preemption and concurrency nests. For example, if a thread nest is aborted, each thread will try to jump to the end of the abortion block,

2. Diagrams synthesized by the SyncChart editor of Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), available at http://www.informatik.uni-kiel.de/rtsys/kieler/

which will be beyond its address range, and hence the thread will be terminated.

5.2 Handling Preemption

According to the Esterel semantics, a preemption (abortion or suspension) is *enabled* when control is in its body, and *disabled* when control is outside of its body. When a preemption is *enabled*, the corresponding trigger signal is watched and the module can react to the presence of it (is *active*). Otherwise, the signal does not cause preemption. We call this scheme *Inside/Outside Preemption Range Watching* (*IOPRW*).

The RC 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 [7]. If during execution of the program the PC is within the watched range and the trigger signal is present, the Watcher triggers the corresponding changes in the control flow. Note that Esterel allows the arbitrary nesting of preemption blocks of different types, for example a strong abortion may be nested within another weak abortion, which may be nested in a suspension. The Reactive Block is responsible for coordinating the Watcher blocks in a way that reflects the Esterel semantics. Each Watcher in the Reactive Block is assigned an index number, which also defines its priority. A Watcher can be overridden by another Watcher with higher priority. Considering the preemption nest structure, it becomes clear that the higher priority preemption has a wider address range which covers all the lower priority ones. Therefore, the earlier preemption instruction in a preemption nest will be assigned to the higher priority Watcher. Note that with this approach, it is not necessary to continuously execute special Checkabort instructions [17] that check on the status of each watcher, meaning that the program does not slow down when entering a (nested) abortion block. The Watcher modules operate autonomously, thus also offering a certain type of concurrency (with true parallelism) beyond the concurrency operator (implemented by interleaved multi-threading).

The KEP 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 KEP includes several versions of the Watcher, with a correspondingly *refined* ISA. 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. The default Watcher is the most general, which can handle all execution contexts.

```
1  module Trap1:
2  output O;
3  % −−−−−−−− Thread 0
4  trap T1 in [
5  % −−−−−−−− Thread 1
6    nothing;
7  ||
8  % −−−−−−−− Thread 2
9      trap T2 in [
10 % −−−−−−−− Thread 3
11       exit T2;
12     ||
13 % −−−−−−−− Thread 4
14       exit T1;
15 % −−−−−−−− Thread 2
16     ] end trap;
17     emit O
18 % −−−−−−−− Thread 0
19 ] end trap;
20 halt;
21 end module
```

(a) Trap1: Thread 2 does not emit O, as exception T1 is thrown within Thread 2

```
1  module Trap2:
2  output O;
3  % −−−−−−−− Thread 0
4  trap T1 in [
5  % −−−−−−−− Thread 1
6    exit T1;
7  ||
8  % −−−−−−−− Thread 2
9      trap T2 in [
10 % −−−−−−−− Thread 3
11       exit T2;
12     ||
13 % −−−−−−−− Thread 4
14       nothing;
15 % −−−−−−−− Thread 2
16     ] end trap;
17     emit O
18 % −−−−−−−− Thread 0
19 ] end trap;
20 halt;
21 end module
```

(b) Trap2: Thread 2 emits O, as exception T1 is thrown by concurrent Thread 1

Fig. 5. Interaction of concurrency and exception handling.

5.3 Handling Exceptions

The KEP does not need an explicit equivalent to the trap statement, but it provides an EXIT statement that specifies a trap scope. 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. This is dictated by the Esterel semantics, which specifies them as a variant of weak aborts.

It is possible that some concurrent threads were executed before executing the EXIT in the current tick and have become inactive. Hence, they cannot directly respond to the exception because they will not be awoken in that tick. Such a situation will be detected at the join point. If there is an active exception, all branch threads that wait at this join point will be set to disabled.

As for trap nests, the question is how to determine which one ought to take priority. The Esterel semantics specifies that outer traps take priority over inner ones. Hence, a simple idea is that the outer trap, which has the larger address range, will override the inner one. Unfortunately, this strategy is too simplistic to satisfy all cases. Compare the Trap1 and Trap2 examples in Fig. 5. As illustrated there, one must not only consider trap nesting structures, but also the concurrency relationships among the threads involved.

To handle this issue, a thread also keeps track of its parent thread. If the PC resides inside of the scope

of a trap, the corresponding exception will be active. When several exceptions are all active, the KEP will compare their parent thread to determine whether they belong to a group of concurrent threads. If they have the same parent thread, the outer trap will cancel the inner one, as in the Trap1 example. Otherwise the Esterel semantics requires to respond to the inner one first, as in Trap2. Furthermore, an inheritance strategy is used when a trap crosses several threads. At the join point, if a thread finds there is an active exception which is emitted by its child thread, it will inherit this exception by adopting the parent thread of this exception as its parent thread, *i.e.*, it will propagate this exception as being emitted by itself. 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: standard delays, immediate delays, and count delays. A delay may elapse in some later tick, or in the same tick in the case of immediate delays. In the KEP, 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.

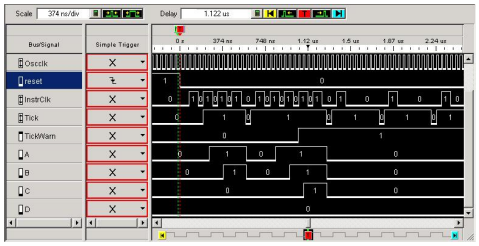
5.5 The Tick Manager and Energy Saving

One of the distinguishing features of the KEP is the Tick Manager. It can autonomously ensure that logical ticks, which for a deployed KEP correspond to one read inputs/compute outputs reaction cycle, are computed at a fixed rate, given a fixed clock frequency. Furthermore, the Tick Manager internally monitors timing violations. The strl2kasm compiler performs a conservative WCRT analysis to determine the tick frequency, and timing violations should never occur [13]. Hence, when using this compiler, the timing violation monitoring can be considered a redundant self-checking run-time mechanism to enhance robustness.

The Tick Manager is activated by setting the pre-defined valued signal _TICKLEN to a certain value. This is typically at the beginning of the program, but may also be done later at run time. The activation of the Tick Manager is optional; if _TICKLEN is not set to any value, the KEP is in "free running" mode and starts computing the next tick as soon as the current one is finished. This will typically be faster on average; however, it will result in a jitter of the reaction time, which is often undesirable from a control perspective.

```
1  INPUT D;
2  OUTPUT
3  A,B,C;
4  EMIT
5   _TICKLEN, #3;
6  EMIT A;
7  EMIT B;
8  PAUSE;
9  EMIT A;
10 EMIT B;
11 EMIT C;
12 AWAIT D;
```

(a) KEP Assembler



(b) Timing diagram, for input D absent; TickWarn indicates a timing violation in the second tick, fourth instruction

Fig. 6. The Tick Manager detects a timing overrun.

As the KEP instruction cycles require a fixed number of clock cycles, providing a value for _TICKLEN can also alleviate the need for the environment to provide a timer that starts the ticks in regular intervals. An external timer is only needed if the clock rate is not stable enough for the application, *e.g.*, due to energy-saving frequency scaling.

If a tick is finished in less than _TICKLEN instruction cycles, the KEP idles for the remaining cycles before starting the next tick. If, on the other hand, a tick is not finished within _TICKLEN cycles, this is considered a *tick length timing violation*. Such timing violations are signaled to the environment via a special signal, TickWarn, with a dedicated output pin; this signal remains present until the next reset of the processor. Furthermore, the self-monitoring makes it easy for the environment to detect any timing violations. The WCRT analysis aims to determine a conservative, yet tight value for _TICKLEN. How a given value for _TICKLEN translates to concrete bounds on physical reaction times also depends on the interface with the environment, as described elsewhere [10].

Fig. 6 illustrates the KEP timing behavior for a small example. In OVERRUN, the first EMIT statement sets _TICKLEN to three; in other words, the module claims that at most three KEP instructions suffice to compute one logical tick. For the input scenario of signal D always absent, the KEP produces the timing shown in Fig. 6b. In this example, the program is running on a KEP implemented on a Memec V2MB1000 Development Board at a rate of $T_{osc} = 41.67ns$, the waveform was recorded by an Agilent 1683A Logic Analyzer. In the example, the first logical tick lasts three instruction cycles. In the second tick, the controller has to execute five instructions until the AWAIT statement is executed. Hence, the Tick Manager will set the TickWarn processor pin high when the fourth instruction cycle is executed to indicate the tick length timing violation.

For controller programming, the main goal of Esterel, the control signals tend to be more often absent than present [14]. The condition of all signals being absent

is called a *blank event*. Even though Esterel does allow to specify reactions for signal absence, typically very few instruction cycles are required for executing a blank event (see also Sec. 6.5). To make the KEP benefit from this when less than _TICKLEN instructions have been executed and there are no instructions needed for the current tick, *i.e.*, all threads are inactive, an IDLE signal will be broadcast to gate the clock of other elements for power reduction [32].

6 Validation and Experimental Results

To validate the correctness of the KEP and its compiler and to evaluate its performance, we employ an evaluation platform whose structure is shown in Fig. 7. The user interacts via a host work station with an FPGA board, which contains the KEP 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 KEP and reads out the generated output events. This also yields the number of instructions per tick, from which we can deduce the WCRT for the given trace. This measures performance, and allows to validate strl2kasm's WCRT analysis with respect to its safety (never underestimates) and accuracy (as little overestimates as possible). 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 input/output 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 KEP and its compiler. The regression test suite currently includes well over 400 Esterel programs.

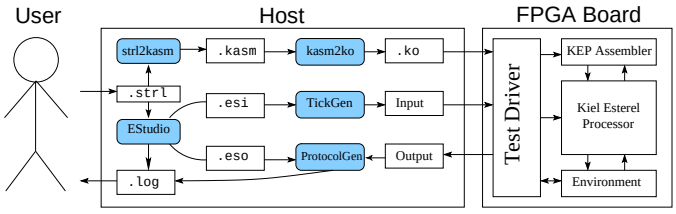


Fig. 7. The KEP evaluation platform.

6.1 Concurrency Analysis

To evaluate the performance of the KEP, we selected eleven standard test cases, from the Estbench[3] suite and other sources [5], [33]. These benchmarks are typical Esterel applications, which not only contain reactive statements, but also include arithmetic and logical data handling. However, we leave out programs that make use of the pre operator, since the CEC currently does not support it [4].

3. http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz

Module Name	Esterel source LOC	Threads Count	Threads Max depth	Threads Max conc.	KEP Code size (words)	KEP Dep. count	KEP CKAG Max priority	KEP CKAG PRIO instr's
abcd	160	4	2	4	164	36	3	30
abcdef	236	6	2	6	244	90	3	48
eight_but	312	8	2	8	324	168	3	66
chan_prot	42	5	3	4	62	4	2	10
reactor_ctrl	27	3	2	3	34	5	1	0
runner	31	2	2	2	27	0	1	0
example	20	2	2	2	28	2	3	6
ww_button	76	13	3	4	95	0	1	0
greycounter	143	17	3	13	343	53	6	58
tcint	355	39	5	17	379	65	3	20
mca200	3090	59	5	49	8650	129	11	190

TABLE 3
Concurrency analysis of benchmarks.

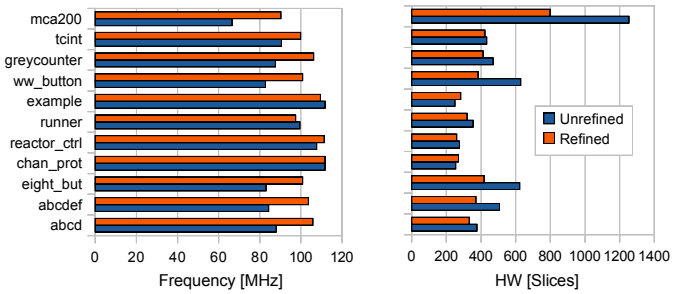


Fig. 8. Clock frequencies and hardware costs with and without ISA refinement.

To characterize each benchmark with respect to its use of concurrency constructs, Table 3 lists the counts and depths of them. For the KEP, the table shows the number of dependencies found, the used number of priority levels (the KEP provides up to 255), and the number of used PRIO instructions. In most cases, the maximum priority used is three or less, indicating relatively few priority changes per tick. For example, eight_buttons has 168 dependencies, but the maximum priority used is 3. On the other hand, greycounter, with 53 dependencies, requires a maximum priority of 6.

6.2 Preemption Analysis and Watcher Comparison

Typically, the preemption constructs tend to be sequential or concurrent rather than being nested. For example, the mca200 employs 64 preemption statements, however, the maximum depth of the preemption nest is just 4. As it turns out, most of the preemptions can be handled by the cheapest Watcher type, the Thread Watcher.

To assess the savings of watcher refinement, we synthesized different variants of the Reactive Core for each benchmark, with and without watcher refinement, see

Module Name	Esterel LOC [1]	MicroBlaze Code+Data (b) V5 [2]	MicroBlaze Code+Data (b) V7 [2]	MicroBlaze Code+Data (b) CEC [2]	KEP Code (w) abs. [3]	KEP Code (w) rel. [3]/[1]	KEP Code+Data (b) abs. [4]	KEP Code+Data (b) rel. [4]/[2]
abcd	160	**6680**	7928	7212	164	1.03	738	0.11
abcdef	236	9352	9624	**9220**	244	1.03	1098	0.12
eight_but	312	12016	**11276**	11948	324	1.04	1458	0.13
chan_prot	42	3808	6204	**3364**	62	1.48	279	0.08
reactor_ctrl	27	2668	5504	**2460**	34	1.26	153	0.06
runner	31	3140	5940	**2824**	27	0.87	121	0.04
example	20	2480	5196	**2344**	28	1.4	126	0.05
ww_button	76	6112	7384	**5980**	95	1.25	427	0.07
greycounter	143	**7612**	7936	8688	343	2.4	1549	0.2
tcint	355	14860	11376	15340	379	1.07	1707	0.15
mca200	3090	104536	77112	**52998**	8650	2.8	39717	0.75

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

Fig. 8. In most cases, having refined watcher types available uses less resources and allows to increase the frequency, and its benefits increase with the scale of the modules. For the industry size mca200 benchmark, refined watchers reduce hardware usage by 36%, and raise the maximum frequency also by 36%. Another benefit of the refined preemption handling architecture is that it keeps the performance stable. If there are no refined watcher types, there is a 40% gap between the highest (112MHz) and the lowest (66MHz) frequency. With refined watchers available, the frequency only degrades by about 19% (from 112MHz to 90MHz).

6.3 Memory Usage

Table 4 compares executable code size and RAM usage between the KEP and the MicroBlaze implementations. For the MicroBlaze, we used three different Esterel compilers (V5, V7, and CEC), and compared ourselves to the best of these. To assess the size of the KEP code related to the Esterel source, we compare the code size in words with the Esterel Lines of Code (LOC, before dismantling, without comments). We notice that the KEP 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 KEP code is typically an order of magnitude smaller than the MicroBlaze code. Furthermore, the more compact state encoding reduces the data memory requirements. The KEP 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. As for the mca200, the memory reduction of the KEP implementation is not so dramatic as that of other cases. The reason is that the mca200 contains lots of data handling—which is not a very strong point of the KEP.

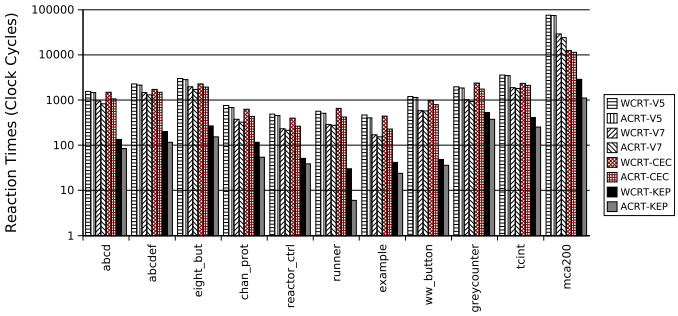


Fig. 9. The worst-/average-case reaction times (in clock cycles) for the KEP and MicroBlaze implementations. Note the logarithmic scale.

6.4 Execution Times

The improvement in execution time of the KEP implementation is shown in Fig. 9. Compared with the best result of the MicroBlaze implementations, the KEP typically obtains more than 4x speedup for the WCRT, and more than 5x for the Average Case Reation Time (ACRT). For a fair comparison, the time is measured based on the system clock. If the comparison is based on the instruction cycles, the KEP will achieve 12x speedup for the WCRT and more than 15x for the ACRT.

The MicroBlaze uses several levels of memory. Here we employed an FPGA chip with a large on-chip memory to implement the MicroBlaze system. Hence, all of the MicroBlaze programs could be loaded into the on-chip memory to make sure that the memory access time is minimal. The MicroBlaze implementation benefits from this, because if the implementation is based on an FPGA which has smaller scale on-chip memory, the KEP program is still likely to fit into the on-chip memory.

6.5 Power Usage

To compare the energy consumptions, we choose the Xilinx 3S200-4ft256 as FPGA platform. This requires 37mW as quiescent power for the chip itself. The MicroBlaze is assumed to run at 50MHz, and the peak power of the MicroBlaze is calculated by the frequency and the hardware resources of the MicroBlaze system via Xilinx WebPower Version 8.1.01. Based on the findings presented in Fig. 9, we calculate the minimal clock frequencies of the KEP to achieve the same WCRT of corresponding MicroBlaze for each benchmark, then calculate the peak power of the KEP implementation.

For most blank events, the action of an Esterel module is very simple—it tests the presence of awaited signals, and then finishes this tick because those await statements are not terminated. For the KEP, since the elapsed instruction cycle count for those actions is far from the assigned tick length, the system will turn to the idle state for saving power. Although the MicroBlaze has no low-power operating mode that can be used to conserve processor energy (*e.g.*, like the wait-state of the

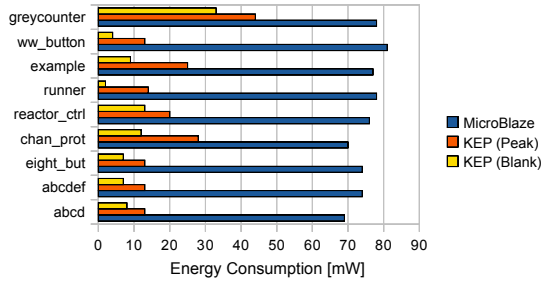


Fig. 10. Energy consumption of KEP and MicroBlaze.

PowerPC405), we still assume it can use some additional circuit to manage its power usage by blocking its clock to satisfy the fixed tick length feature. Note that the real tick length for a blank event depends on the state of the program of the previous tick. The average power usage of blank events is also estimated by an extended esi file, which inserts a blank event between every two original ticks. Fig. 10 shows that the KEP reduces energy usage on average by 75%. The reduction becomes even more significant for blank events. Energy consumptions of the MicroBlaze system are similar for different events. However, the reactive architecture makes the power usage of the KEP 52% lower than its peak power. Hence, in this case, the KEP achieves 86% power savings.

7 Conclusions & Outlook

We have presented the KEP, a multi-threaded processor, which allows the efficient, predictable execution of concurrent Esterel programs. 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. An Esterel description of the KEP is available as open source[4].

To accurately capture the Esterel semantics in a reactive processing setting is not trivial, as has become evident from earlier (failed) attempts. So far, we are relying on extensive experimental validation to ensure correctness, as described in Sec. 6; a formal treatment of the reactive processing approach is underway [34].

It would be interesting to implement a virtual machine that has an instruction set similar to the KEP; see also the recent proposal by Plummer *et al.* [35]. Furthermore, the underlying model of computation, with threads keeping their individual program counters and a priority based scheduling, can also be emulated by classical processors. In fact, the recent SyncCharts in C (SC) proposal [27] defines a set of operators for reactive control flow that follow the KEP's execution model quite closely. The main differences are that SC does not support traps (a simplification of SyncCharts compared to Esterel) and that the thread id/priority mechanism is reduced to only ids, which serve as priorities as well. Interestingly, the current reference implementation for SC[5] can take advantage of machine instructions for arithmetic operations that are typically not available at the program level. Specifically, on the x86, the scheduler uses a bit vector to represent active threads and the bsr (Bit Scan Reverse) assembler instruction to determine the active thread with the highest id.

The SC operators are directly embedded in regular C, no prior compilation or OS support is necessary; hence, in a way, SC uses the ISA of a traditional processor in a reactive processing manner. At least in terms of code compactness, this approach could be competitive with the custom reactive processing approach; to what extent it can match reactive processing in terms of performance, power consumption and predictability is still open for investigation.

Acknowledgments

This work has benefited from discussions with many people, in particular Michael Mendler and Stephen Edwards. We would also like to thank Marian Boldt for developing the strl2kasm compiler, Özgün Bayramoglu and Hauke Fuhrmann for the KIELER visualization, and the reviewers for providing very valuable feedback on this manuscript.

References

[1] G. Berry, "Esterel on Hardware," *Philosophical Transactions of the Royal Society of London*, vol. 339, pp. 87–104, 1992.

[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[3] S. A. Edwards, "High-Level Synthesis from the Synchronous Language Esterel," *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, Jun. 2002.

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

[5] 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.

[6] Z. A. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli, "REFLIX: A processor core for reactive embedded applications," in *Proceedings of the 12th International Conference on Filed Programmable Logic and Applications (FPL-02)*, ser. Lecture Notes in Computer Science, M. Glesner, P. Zipf, and M. Renovell, Eds., vol. 2438. Montpellier, France: Springer, Sep. 2002, pp. 945–945.

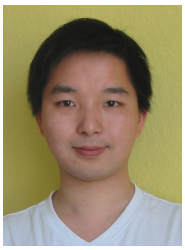
[7] X. Li and R. von Hanxleden, "A concurrent reactive Esterel processor based on multi-threading," in *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[8] X. Li, M. Boldt, and R. von Hanxleden, "Mapping Esterel onto a multi-threaded embedded processor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.

4. http://www.informatik.uni-kiel.de/rtsys/kep/

5. http://www.informatik.uni-kiel.de/rtsys/sc/

[9] S. Yuan, S. Andalam, L. H. Yoong, P. S. 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, Apr. 2008.
[10] 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 the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*. San Francisco, CA, USA: ACM Press, Sep. 2005, pp. 225–236.
[11] X. Li, "The Kiel Esterel Processor: A multi-threaded reactive processor," Ph.D. dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Jul. 2007, http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_00002198.
[12] M. Boldt, "A compiler for the Kiel Esterel Processor," Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2007, http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf.
[13] M. Boldt, C. Traulsen, and R. von Hanxleden, "Worst case reaction time analysis of concurrent reactive programs," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 4, pp. 65–79, Jun. 2008, proceedings of the International Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'07), March 2007, Braga, Portugal.
[14] G. Berry, *The Esterel v5 Language Primer, Version v5_91*, Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000, ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf.
[15] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, May 2007.
[16] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong, "Reactive processing for reactive systems," *ERCIM News*, vol. 67, pp. 28–29, Oct. 2006.
[17] 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, Sep. 2004.
[18] X. Li and R. von Hanxleden, "The Kiel Esterel Processor - a semi-custom, configurable reactive processor," in *Synchronous Programming - SYNCHRON'04*, ser. Dagstuhl Seminar Proceedings, S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner, Eds., no. 04491. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005, http://drops.dagstuhl.de/opus/volltexte/2005/159.
[19] Z. A. Salcic, D. Hui, P. S. Roop, and M. Biglari-Abhari, "REMIC: Design of a reactive embedded microprocessor core." in *Proceedings of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Shanghai, China, 2005, pp. 977–981.
[20] M. W. S. Dayaratne, P. S. Roop, and Z. Salcic, "Direct Execution of Esterel Using Reactive Microprocessors," in *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, Apr. 2005.
[21] L. H. Yoong, P. Roop, and Z. Salcic, "Compiling Esterel for distributed execution," in *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, Apr. 2006.
[22] S. Gädtke, C. Traulsen, and R. von Hanxleden, "HW/SW Co-Design for Esterel Processing," in *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS'07)*, Salzburg, Austria, Sep. 2007.
[23] C. Traulsen and R. von Hanxleden, "Reactive parallel processing for synchronous dataflow," in *Proceedings of the 25th Symposium On Applied Computing (SAC'10), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Sierre, Switzerland, Mar. 2010.
[24] S. A. Edwards and E. A. Lee, "The case for the Precision Timed (PRET) machine," in *Proceedings of the 44th Design Automation Conference*, San Diego, CA, USA, Jun. 2007, pp. 264–265.
[25] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen, "Tight WCRT analysis for synchronous C programs," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, Grenoble, France, Oct. 2009.
[26] M. Mendler and M. Pouzet, "Uniform and modular composition of data-flow & control-flow in the lazy λ-calculus," Presentation at the International Open Workshop on Synchronous Programming (SYNCHRON'08), Aussois, France, Dec. 2008.
[27] R. von Hanxleden, "SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency," in *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, Oct. 2009.
[28] X. Li and R. von Hanxleden, "KEP2 (Kiel Esterel Processor 2): The Esterel Processor," Christian-Albrechts-Universität Kiel, Department of Computer Science, Technical Report 0506, Apr. 2005.
[29] J. Lukoschus and R. von Hanxleden, "Removing cycles in Esterel programs," *EURASIP Journal on Embedded Systems, Special Issue on Synchronous Paradigms in Embedded Systems*, 2007, http://www.hindawi.com/getarticle.aspx?doi=10.1155/2007/48979.
[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987, http://doi.acm.org/10.1145/24039.24041.
[31] C. André, "SyncCharts: A visual representation of reactive behaviors," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–52, rev. RR 96–56, Rev. April 1996, http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf.
[32] L. Benini, A. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," in *Readings in hardware/software co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002, pp. 231–248.
[33] G. Berry, *The Esterel v5 Language Primer*, 1999, ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps.
[34] C. Traulsen, "The Kiel Reactive Processor (KReP)," Ph.D. dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, in preparation.
[35] B. Plummer, M. Khajanchi, and S. A. Edwards, "An Esterel virtual machine for embedded systems," in *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, Mar. 2006.



Xin Li obtained his B. S. in Measurement Technology and Instrument Design in 1996 and his M. Sc. in Mechanical and Electronic Engineering in 2000, both from Wuhan University of Technology, P. R. China. Until 2003 he was lecturer in the School of Mechanical and Electronic Engineering at Wuhan. He completed his Ph. D. at the Dept. of Computer Science of Christian-Albrechts-Universitä Kiel, in 2007. He now is with the Laboratory for Advanced Research in Computing Technology and Compilers in the Dept. of Electrical and Computer Engineering at the University of Minnesota. His current research interests focus on embedded system design and reactive processing. He holds two patents.

Reinhard von Hanxleden studied Computer Science and Physics at the Christian-Albrechts-Universität (CAU) Kiel from 1985 to 1988 and completed his M. Sc. in CS at The Pennsylvania State University in 1989. He performed his doctoral studies at the CS Dept./Center for Research on Parallel Computation at Rice University in the field of data-parallel compilation until 1994, with the Ph. D. conferred in 1995. He subsequently joined Daimler Chrysler research, until 2000 with the Responsive Systems in Berlin, afterwards with Airbus in Hamburg and Toulouse. He joined the CAU CS faculty in 2001 as head of the real-time/embedded systems group. His interests include model-based design, concurrency and synchronous programming. He is currently involved in the IEEE standardization of the Esterel language and is a member of ACM, IEEE, and GI.