HW/SW Co-Design for Esterel Processing

Sascha Gädtke, Claus Traulsen, and Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Dept. of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{sga,ctr,rvh}@informatik.uni-kiel.de

ABSTRACT

We present a co-synthesis approach that accelerates reactive software processing by moving the calculation of complex expressions into external combinational hardware. The starting point is a system model written in the synchronous language Esterel, which can be mapped to both hardware and software. Our approach performs the partitioning at the source-code level and preserves the original, strictly synchronous semantics. It is thus platform-independent and allows to use standard simulation and synthesis tools. Furthermore, the source-level partitioning approach presented here should be applicable to non-reactive processing platforms as well. However, the challenge is to partition the program without changing its meaning under any circumstances. In particular, signal scopes and inter-partition signal dependencies must be maintained, which rules out a naïve top-level partitioning.

We have implemented the co-synthesis approach based on the Columbia Esterel Compiler and have validated it on the Kiel Esterel Processor. As the experimental results confirm, this can significantly reduce execution times and energy consumption per reaction, with minimal additional hardware requirements.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms

Design, Performance

Keywords

HW/SW Co-Design, Synchronous Languages, Esterel, Reactive Processing

1. INTRODUCTION

Most embedded systems are reactive, *i. e.*, they must react continuously to stimuli from their physical environment, with the speed that is imposed by the environment. The requirements for these systems differ from usual software systems. Raw processing power is typically less important than deterministic functional behavior and predictable timing. Since the physical environment behaves in parallel, they often have to run concurrent tasks, each of which has hard deadlines. In order to allow deterministic behavior, the threads should be statically schedulable and the execution times should have low jitter. Low power consumption is mandatory for battery-powered embedded systems.

To deal with these specific requirements synchronous languages [2] were designed, especially to support preemption and concurrency. They rely on the *synchrony hypothesis*, which assumes that the computation takes no time, or, more practically, the computation time is small compared to the time between significant changes of the input. This assumption allows to define a clear formal semantics for these languages.

An imperative synchronous language is ESTEREL [4]. An ESTEREL program interacts internally and with the environment through *signals*. The execution is divided into multiple, discrete *(logical) ticks* or *instants*, where signals are considered constant during one tick. ESTEREL supports different types of preemption as well as deterministic concurrency. ESTEREL programs can be compiled to VHDL for hardware generation, or they can be executed in software. The latter traditionally involves the synthesis of a C program, which is then executed on a regular processor. An alternative, more recent software synthesis approach employs *reactive processors*, which are specifically designed to execute reactive programs efficiently and with predictable timing [14]. Unlike traditional processors, they directly support preemption and concurrency. However, like other processors, they are inefficient for computing complex expressions relative to a hardware implementation.

Reactive processors can very efficiently test for the presence of specific signals and change control flow accordingly. However, ESTEREL also permits the usage of *signal expressions*, which are arbitrary boolean combinations of signals, and here any software-based approach, including reactive processors, must perform explicit, time-consuming computations. The aim of the work presented here is to accelerate software implementations of ESTEREL programs by delegating the computation of complex expressions into external combinational hardware. Further objectives were to maintain a strictly semantics, to require minimal modifications of the reactive processing platform, and to still allow the use of standard simulation and synthesis tools.

The main contribution of the paper is a source-level partitioning and co-synthesis approach that, unlike earlier co-design approaches based on ESTEREL, fully preserves the original synchronous ESTEREL semantics. Source-level partitioning offers the advantage of being platform-independent and allows to use standard simulation and synthesis tools. Furthermore, the partitioning approach should be applicable to non-reactive processing platforms as well.

We have validated our approach by extending a reactive processor, the *Kiel Esterel Processor* (KEP), by an additional logic block that computes signal expressions. Signal expressions are extracted automatically from an ESTEREL program and translated into hardware. The results are fed as additional inputs into the *Kiel Esterel Processor* (KEP), which executes a modified program version that tests these additional inputs instead of computing the signal expressions explicitly. The behavior of the transformed program and the original program do not differ, except that the transformed program is accelerated by the elimination of expression computations. As the external combinational hardware does not affect the critical path of the processor, the maximal clock rate is not reduced. As the experimental results indicate, this co-synthesis approach lowers overall energy consumption per tick and can also decrease the *Worst Case Reaction Time* (WCRT).

In the remainder of this section, we give a short overview on ESTEREL and the KEP and consider related work. In Section 2 we describe the co-synthesis approach in further detail. Experimental results are presented in Section 3, the paper concludes in Section 4.

1.1 Esterel

The concurrent synchronous programming language ESTEREL is tailored for implementing control-dominated reactive systems. In the synchronous programming model the execution of an ESTEREL program is divided into logical *ticks*, where the execution of one tick is conceptually instantaneous. ESTEREL's signals take exactly one state per tick: *present* or *absent*. Input signals are sampled at the beginning of a tick and output signals are written at the end of it. The status of a signal S can be tested by present S and set to present by emit S for one instant and sustain S for all following instants. Signals can carry additional *values*, such as integers or booleans, which are constant during one tick. In contrast, simple variables may take multiple values. The status and value of a signal S in the previous tick can be tested by pre(S) and pre(?S), respectively.

The key statements of ESTEREL are: wait until a signal expression becomes true (await E) and abortion and suspension of a program-part (abort p when E, suspend p when E). Furthermore, it is possible to execute program parts in parallel, combined by the || operator. Causality cycles, where the status of a signal dynamically depends on itself, are forbidden and detected at compile time. ESTEREL programs are structured by *modules*, where each module has a set of input and output signals; a module may be instantiated in another module.

A simple ESTEREL program is shown at the top of Fig. 1. The inputs A and B are pure signals, while D, E, F carry additional values. In the initial tick, the program checks whether one of the input signals A or B and the local signal C are present; if this is the case, it emits output signal O1. (For this program, the local signal C is not emitted, hence O1 is never emitted either.) Similarly, O2 is emitted depending on D, E, and F. This example program terminates *instantaneously*, meaning that it completes within the initial tick.

1.2 The Kiel Esterel Processor

The upper-left part of Fig. 1 shows the *KEP Assembler* (KASM) code for a SW-only synthesis variant. The statement present (A or B) and C in line 10 of the example ESTEREL program is decomposed into the KASM instructions in lines 13–16. If signal C is absent, the whole signal expression is false, and PRESENT C, A0 (line 13) will perform a conditional jump to A0. If C is present, we have to evaluate A or B (lines 14–16). If for example A is *present*, then the following GOTO A1 instruction is executed and O1 will be emitted. The condition if ((?D<5) or (?E and ?F)) is calculated in lines 18–27. As a first step, in line 18 the value of signal D is loaded into register REG0. The instruction CMP REG0,#1 compares the register with the numerical constant for true. JW EE, A5 is a conditional jump for values. If the last comparison was true, GOTO A4 is executed and finally O2 with value 1 (*true*) will be emitted. Otherwise the program jumps to A5. The remaining instructions are executed accordingly.

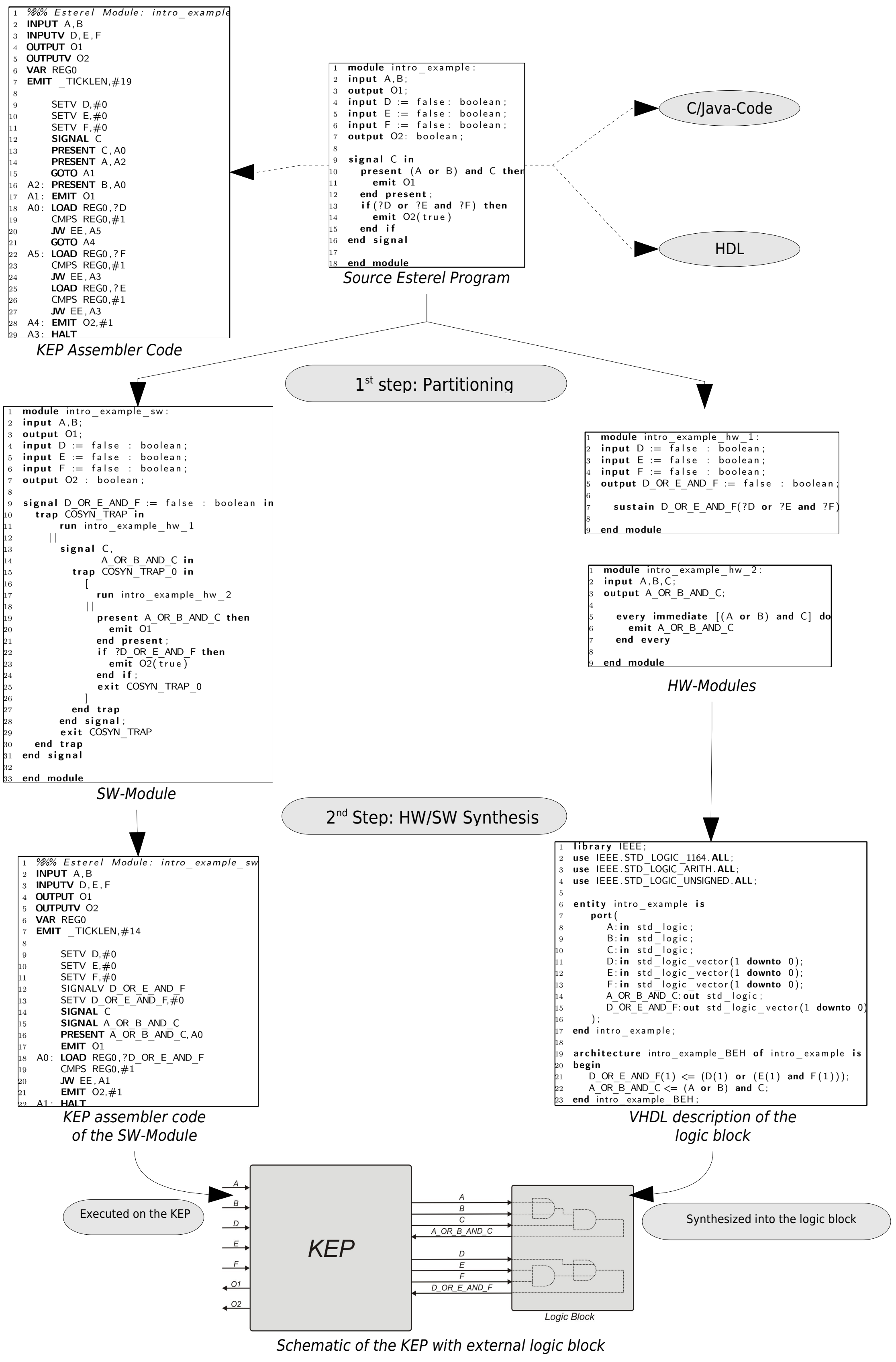
As mentioned before, one of the characteristics of reactive processors is their timing predictability, which results from the direct mapping from the synchronous behavioral description to the instruction set architecture. This makes it feasible to analytically derive conservative, yet fairly accurate WCRT estimates, which give an upper bound on how many instructions the computation of one logical tick can consume. Per default, the KEP reacts as fast as possible, which may cause a jitter of the reaction time. Alternatively, the KEP allows to specify a maximum number t of instruction cycles for a logical tick. This is done by initializing a *tick manager*, by emitting t on the reserved valued signal _TICKLEN. If a tick completes before t instruction cycles have been executed, the execution is stalled until t instruction cycles have passed, in order to reduce the jitter of the reaction time. This results in a constant reaction time, which is often desirable for stable control. For the example, the WCRT analysis built into the KEP compiler has computed that at most $t = 19$ instructions will be needed per tick, line 7 of the KASM code initializes the tick manager accordingly.

As the example illustrates, complex signal and valued expressions have to be computed with numerous KASM instruction, and it might even be necessary to introduce new auxiliary signals. This is likely to increase the WCRT and the resource usage. On the other hand, the types of expressions that appear to be most common—boolean functions and value comparisons—can be computed efficiently in combinational hardware. Therefore, we propose to compute signal and valued expressions in an external logic block wired to the reactive processor.

1.3 Related Work

The work presented here links two areas, namely compilation of Esterel programs and HW/SW co-design, which are discussed in turn. As previously introduced, the most common ways for executing ESTEREL programs are hardware and software synthesis [3, 7]. This is supported by established compilers like the *Columbia Esterel Compiler* (CEC) [6] or the *Esterel V5* compiler from Esterel Technologies. Another way to run ESTEREL programs are reactive processors. The first generations only permitted sequential programs. Later versions, such as the EMPEROR [12] or the KEP3 [9] also permit concurrency. The reactive processing approach aims to make reactive programs more efficient by providing a well-adapted instruction set architecture; our approach complements this by providing dedicated logic that supplements signal-testing instructions.

There have been numerous approaches to perform HW/SW co-design, where a (high-level) system description is somehow partitioned into hardware and software [10]. The POLIS project [1] is based on Esterel as description language, and builds on an execution platform consisting of a micro controller combined with a user-specific hardware block. The ESTEREL program is divided into a software part, a hardware part and an interface in between. The HW and SW parts are identified manually or by a fairly complex performance analysis, whereas our partitioning approach is purely structural. The POLIS system allows an arbitrary partitioning, but this comes at the expense of departing from Esterel's strictly synchronous semantics in favor of a globally asynchronous, locally synchronous (GALS) semantics. We restrict hardware partitions to those that are computable within one tick, which makes it feasible to preserve strict synchrony.


C/Java-Code
HDL
KEP Assembler Code
Source Esterel Program
1st step: Partitioning
SW-Module
HW-Modules
2nd Step: HW/SW Synthesis
KEP assembler code of the SW-Module
VHDL description of the logic block
Executed on the KEP
Synthesized into the logic block
KEP
A
B
D
E
F
O1
O2
C
A_OR_B_AND_C
D_OR_E_AND_F
Logic Block
Schematic of the KEP with external logic block

```
%%% Esterel Module: intro_example
INPUT A,B
INPUTV D,E,F
OUTPUT O1
OUTPUTV O2
VAR REG0
EMIT _TICKLEN,#19

        SETV D,#0
        SETV E,#0
        SETV F,#0
        SIGNAL C
        PRESENT C,A0
        PRESENT A,A2
        GOTO A1
A2:     PRESENT B,A0
A1:     EMIT O1
A0:     LOAD REG0,?D
        CMPS REG0,#1
        JW EE,A5
        GOTO A4
A5:     LOAD REG0,?F
        CMPS REG0,#1
        JW EE,A3
        LOAD REG0,?E
        CMPS REG0,#1
        JW EE,A3
A4:     EMIT O2,#1
A3:     HALT
```

```
module intro_example:
input A,B;
output O1;
input D := false: boolean;
input E := false: boolean;
input F := false: boolean;
output O2: boolean;

signal C in
  present (A or B) and C then
    emit O1
  end present;
  if(?D or ?E and ?F) then
    emit O2(true)
  end if
end signal

end module
```

```
module intro_example_sw:
input A,B;
output O1;
input D := false : boolean;
input E := false : boolean;
input F := false : boolean;
output O2 : boolean;

signal D_OR_E_AND_F := false : boolean in
  trap COSYN_TRAP in
      run intro_example_hw_1
    ||
      signal C,
             A_OR_B_AND_C in
        trap COSYN_TRAP_0 in
          [
            run intro_example_hw_2
          ||
            present A_OR_B_AND_C then
              emit O1
            end present;
            if ?D_OR_E_AND_F then
              emit O2(true)
            end if;
            exit COSYN_TRAP_0
          ]
        end trap
      end signal;
      exit COSYN_TRAP
  end trap
end signal

end module
```

```
module intro_example_hw_1:
input D := false : boolean;
input E := false : boolean;
input F := false : boolean;
output D_OR_E_AND_F := false : boolean;

  sustain D_OR_E_AND_F(?D or ?E and ?F)

end module
```

```
module intro_example_hw_2:
input A,B,C;
output A_OR_B_AND_C;

  every immediate [(A or B) and C] do
    emit A_OR_B_AND_C
  end every

end module
```

```
%%% Esterel Module: intro_example_sw
INPUT A,B
INPUTV D,E,F
OUTPUT O1
OUTPUTV O2
VAR REG0
EMIT _TICKLEN,#14

        SETV D,#0
        SETV E,#0
        SETV F,#0
        SIGNALV D_OR_E_AND_F
        SETV D_OR_E_AND_F,#0
        SIGNAL C
        SIGNAL A_OR_B_AND_C
        PRESENT A_OR_B_AND_C,A0
        EMIT O1
A0:     LOAD REG0,?D_OR_E_AND_F
        CMPS REG0,#1
        JW EE,A1
        EMIT O2,#1
A1:     HALT
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity intro_example is
    port(
        A:in std_logic;
        B:in std_logic;
        C:in std_logic;
        D:in std_logic_vector(1 downto 0);
        E:in std_logic_vector(1 downto 0);
        F:in std_logic_vector(1 downto 0);
        A_OR_B_AND_C:out std_logic;
        D_OR_E_AND_F:out std_logic_vector(1 downto 0)
    );
end intro_example;

architecture intro_example_BEH of intro_example is
begin
    D_OR_E_AND_F(1) <= (D(1) or (E(1) and F(1)));
    A_OR_B_AND_C <= (A or B) and C;
end intro_example_BEH;
```

Figure 1: Overview of Esterel synthesis paths. Dashed lines indicate the traditional, software-only (KEP Assembler or C/Java) or hardware-only (HDL) synthesis alternatives. Solid lines indicate the co-synthesis flow presented here; the first step partitions the ESTEREL program into one software module and an arbitrary number of hardware modules, the second step compiles these to KEP assembler and VHDL, respectively. Note that the schematic at the bottom does not match the example, it instead illustrates how the resource usage in the logic block is minimized by reusing subexpressions.

2. HW/SW CO-SYNTHESIS

HW/SW co-synthesis of an ESTEREL program for the KEP is performed in the following steps.

Step 1: Partition the ESTEREL program into software and hardware modules on the source code level.

Step 2a: Compile the software module to KASM, for execution on the KEP.

Step 2b: Compile the hardware modules to a VHDL description of a combinational logic, which is interfaced to the KEP.

Step 3: Provide the interface between KEP and combinational logic.

We have implemented the complete co-synthesis design flow for the KEP, based on the CEC compiler infrastructure. Fig. 1 illustrates the resulting HW and SW modules for the example. The individual synthesis steps are discussed in more detail in the following.

2.1 Step 1: HW/SW Partitioning

The ESTEREL program is divided into a software module and hardware modules. The SW module is similar to the original, except that complex (non-atomic) expressions are replaced by fresh auxiliary signals. The HW modules are responsible for computing these auxiliary signals. As noted before, we want to do this partitioning on the Esterel-level, *e. g.*, to be able to do a co-simulation, and to allow the usage of standard SW and HW synthesis tools downstream. While the basic source-level partitioning approach is relatively straightforward, there are two issues that we have to be aware of, namely to respect *signal scoping* and *signal dependencies*. The scoping issue will be addressed in the sequel, the dependency issue will be addressed in Section 2.2.

A first approach to do the partitioning would be to generate one HW module that continuously computes all necessary auxiliary signals, and to run this in parallel to the SW module. However, this would not respect the scoping of local signals, which may be part of a signal expression. While the status and value of a signal is constant within one tick, the scope of a local signal may be entered multiple times, due to loops. The signal is considered as a new signal each time, hence it can have different values. This behavior is called *schizophrenia* [4], and our logic has to preserve this behavior. There are means to eliminate schizophrenia, at the Esterel level [13]. Hence one option to handle local signals and schizophrenia in our partitioning would be to first apply one of the known schizophrenia elimination schemes, and then to declare all local signals at the outermost level. However, this approach would come at the cost of a certain potential increase in code-size, also effecting the size of the resulting KASM program.

As a more efficient alternative to source-level elimination of schizophrenia, we do not generate just one HW module, to be combined with the SW module at the outermost level, but instead generate a hardware module for each signal scope, which runs parallel to the body of the signal scope. The auxiliary signal for one signal expression is calculated in the HW module corresponding to the outermost signal scope (if nested), where all signals are known. All signals of the signal expression are declared as inputs to this HW module and the auxiliary signal is an output. Just like a signal may be incarnated several times within a tick, the HW module may be activated several times within a tick, and may compute different values for its output signals within a tick. However, this does not pose a problem for the reactive co-design implementation. The key is that the HW module is executed not just once per logical tick, but is executed during each instruction cycle, and the different incarnations correspond to separate instruction cycles.

Consider the signal expression in line 10 of the ESTEREL program in Fig. 1. The expression contains both the interface signals A and B and a local signal C. The signal expression in the SW module is substituted by the auxiliary signal A_and_B_or_C. The auxiliary signal is calculated in the HW module 2 corresponding to the local signal scope of C. It is emitted by the emit statement inside the every loop, whenever the signal expression is true. If there is more than one auxiliary signal to be calculated within the HW module, multiple every statements are run in parallel. In the SW module, the HW module is run parallel to the original body of the signal S in ... statement. Additionally the new body is embedded in a trap statement, which terminates the infinitely running HW module when the end of the original body of the signal statement is reached. Note that in this example, the analyzed WCRT is 14 instruction cycles, as opposed to 19 cycles for the SW-only version.

Data Expressions that are, *e. g.*, part of if tests, emissions of valued signals, variable assignments or trap handlers, can also be handled if they contain just boolean signals. If the expression is mixed and also contains comparisons or arithmetics, it is decomposed into its boolean and non-boolean or variable-containing parts. Non-boolean or variable containing subexpressions are substituted with additional auxiliary signals that are emitted immediately before the statement containing the data expression. In this case we have a trade-off between the need for more signals and the reduction of instructions in the KASM code.

Contained pre(S) or pre(?S) operators in an expression are substituted by auxiliary signals with the state and value of S in the previous instance. The KEP provides data structures that carry the state and value of each signal in the previous instance, hence it is possible to substitute the auxiliary signals with these values in further synthesis, and there is no need to use extra signals in HW.

2.2 Step 2a: Software Synthesis

Before compiling the resulting ESTEREL program it would suffice to substitute the bodies of the HW modules with nothing statements, but it is useful to do a post-processing that drops all superfluous statements introduced during partitioning, *i. e.*, the trap and exit, the run and the || statements.

A complication occurs when compiling the SW module to KASM during the calculation of the static scheduling, because dependency information between readers and writers of a signal is lost in the partitioning step. The compiler is not able to reconstruct the information from which signals the newly generated auxiliary signals depend on, thus it cannot be guaranteed that writers of a signal are executed before the readers. To correct this, the compiler stores the signal dependencies of the original program while performing the partitioning, and examines these dependencies them when compiling into KASM.

2.3 Step 2b: Hardware Synthesis

As a pre-processing step for the HW modules, we apply a two-level logic minimization on all expressions used in the HW modules and try to extract good common factors from different expressions. For this, all expressions are exported into the Berkeley Logic Interchange Format (BLIF). Logic minimization is performed with the UC Berkeley package MV-SIS [8]. The minimized BLIF file is then transformed back into appropriate ESTEREL statements.

From the HW modules a VHDL description of the logic block is generated. The logic block is a simple combinational logic; all auxiliary signals get assigned with a boolean composition of signals

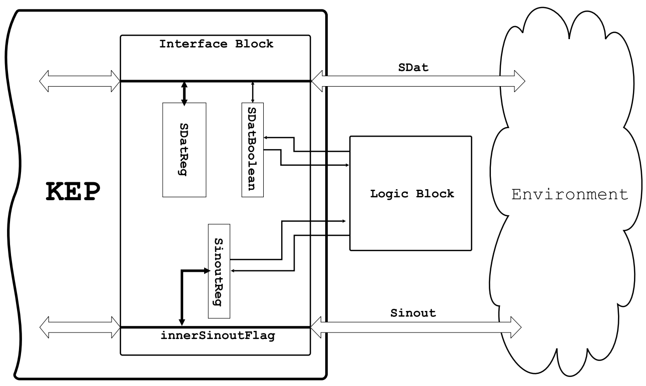


Figure 2: The interface of the logic block to the KEP. The same interface as to the environment is used.

depending on the signal and data expressions in the HW modules. In principle, one might use one of the existing Esterel compilers to synthesize VHDL from the HW modules, but this would impose a large overhead because the every statement would be translated into non-trivial logic. We therefore have developed a custom VHDL generator that directly implements the logic for the expressions.

The logic block is connected to the usual interface of the KEP (Fig. 2). For the KEP the only difference between signals that come from the environment and signals that come from the logic block is that signals from the logic block may change their value within one tick. Since the value of a signal expression is never needed in the same instruction cycle in which one of its signals is written, the correct value is always computed fast enough by the logic block. In particular, there is not need to adjust the clock frequency of the KEP. For the same reason, reincarnation is handled correctly: whenever a signal scope is reentered, the signal is reset, and the new value of a signal expression is computed.

2.4 Step 3: Interface Mapping

The KEP supports a configurable number of *n* signals. The interface to the environment consists of a *n*-bit wide bidirectional data bus to read and write signal states and a port to access the signal values. Because of the bit width of valued signals, values can only be accessed sequentially. For this, the port consists of an address bus and a bidirectional data bus.

Internally, signal states for local and output signals are stored in the SinoutReg register and signal values for all interface and local signals are stored in a 32 bit wide RAM. A bus combines the states of the input signals coming from the Sinout data bus with the states of output and local signals. Both the KEP and the environment access the RAM; while a tick is executed, the RAM is accessed internally by the KEP, and between two ticks the environment gets access to the RAM.

Providing signal states for the logic block as input is simply achieved by connecting the localSinoutFlag, that provides the signal states of all signals, to the logic. The states of the auxiliary signals calculated by the logic are directly fed back to the inputs of the registers. While a RAM just provides sequential access, data must be accessible in parallel for a combinational logic. Therefore, the boolean values for each signal are redundantly stored in the additional SDatBoolean register. The outputs of this register are connected to the logic to provide (boolean) signal values. The calculated auxiliary signals are fed back into the register. There are no writing access conflicts, because the auxiliary signals are only written by the logic; and only the logic writes these signals.

3. EXPERIMENTAL RESULTS

To validate and evaluate the co-synthesis approach presented here, we have run a number of benchmarks on the KEP. The co-synthesis is performed automatically on the benchmark programs, and they are run with a set of input traces, which were generated for full coverage of the original programs.

Obviously, the potential benefit of this approach varies from program to program and depends heavily on how frequently complex expressions are used. We cannot expect to have a benefit for programs without signal expressions, but such programs are already executed efficiently on the KEP. Therefore, we have chosen a set of programs that contain a noticeable number of signal expressions.

The co-design approach is a trade-off between the reduction of the number of KEP instruction cycles per tick and an increased resource usage on the KEP and on the execution platform. In the experiments we compare the following values for the normal execution of an ESTEREL program on the KEP and for the co-design approach.

- **Ticklength** The "WCRT" values are conservative, analytically derived upper bounds the reaction time, used for the determination of _TICKLEN. The "AVE" and "MAX" values are measured average and maximum number of instruction cycles per tick, for input traces generated automatically by Esterel Studio.

- **Signals** This indicates how many auxiliary signals are used to substitute expressions.

- **FPGA Utilization** The number of occupied slices and used 4 input Look Up Tables (LUTs) on the FPGA.

- **Energy** The average energy needed to compute one tick, derived as the power consumption of the KEP, as estimated by Xilinx XPower, multiplied with the time to compute one logical tick, *i. e.*, the number of instructions in a tick and the time to execute one instruction.

We compare the standard KEP design with the co-design for five benchmarks: The TCINT taken from the Estbench [5], an acyclic version of the TOKEN RING ARBITER [11] with 3 and 10 stations, a simple FILTER, and a BACKHOE simulation. The platform for the experiments is a KEP configuration that supports up to 85 signals and 60 threads. In all experiments the hardware (KEP and logic) was synthesized for and executed on a Xilinx Virtex 2 Pro (xc2vp30-6ff896) FPGA and was clocked with a frequency of 40 MHz.

Table 1 shows the experimental results. Compared to normal execution on the KEP, the co-design reduces the tick length between roughly ten and sixty percent. For most benchmarks, the biggest reduction affects the analytically derived WCRT, meaning that the reaction frequency (assuming constant reaction time) can benefit the most. There are also significant differences concerning the trade-off between the advantage of a reduced tick length and the demand for more signals. The FILTER benchmark requires 70% more signals, resulting an a WCRT reduction by one third; in the BACKHOE example, we only need two additional signals (6.7%) to reduce the execution time by nearly one half. The extra resource usage induced by the logic gates needed for the logic block and its connection is low: between 1.4% and 2.1% for the number of slices used on the FPGA and between 1.9% and 2.2% for the number of 4 input look up tables. The energy savings per tick are noticeable in all benchmarks and ranges from 14% to almost 60%.

Benchmark	Design	Ticklength			Signals			FPGA Utilization		Energy
		WCRT	AVE	MAX	In	Out	Local	# slices	# 4 input LUTs	(μWs/Tick)
TCINT	KEP only	155	59	101	19	20	12	5144	7591	7.237
	Co-Design	115	51	89	19	20	25	5247	7751	5.591
	Diff	-40	-8	-12		+13		+103	+160	-1.646
	% Diff	-25.8	-13.6	-11.9		+25.5		+2.0	+2.1	-22.7
TOKEN RING ARBITER (3 stations)	KEP only	74	51	61	3	3	16	5144	7591	4.051
	Co-Design	61	46	50	3	3	19	5254	7735	3.469
	Diff	-13	-5	-11		+3		+110	+144	-0.582
	% Diff	-17.6	-9.8	-18.0		+13.6		+2.1	+1.9	-14.4
TOKEN RING ARBITER (10 stations)	KEP only	256	172	201	10	10	51	5144	7591	13.743
	Co-Design	208	148	155	10	10	61	5214	7762	11.460
	Diff	-48	-24	-46		+10		+70	+171	-2.284
	% Diff	-18.75	-14.0	-22.9		+19.6		+1.4	+2.3	-16.6
FILTER	KEP only	133	94	101	16	7	14	5144	7591	7.286
	Co-Design	90	77	83	16	7	40	5217	7762	5.322
	Diff	-43	-17	-18		+26		+73	+170	-1.964
	% Diff	-32.3	-18.1	-17.8		+70.2		+1.4	+2.2	-27.0
BACKHOE	KEP only	209	16	32	12	15	3	5144	7591	8.171
	Co-Design	86	9	22	12	15	5	5254	7734	3.465
	Diff	-123	-7	-10		+2		+110	+143	-4.707
	% Diff	-58.9	-43.7	-31.2		+6.7		+2.1	+1.9	-57.6

Table 1: Experimental Results.

4. CONCLUSION AND FURTHER WORK

We have shown how to extract signal expressions from ESTEREL programs, without changing the semantics. This can be generalized to extracting arbitrary instantaneous code parts, in order to execute them on a more efficient platform. We have chosen signal expressions, because the KEP—like any software processing approach—is not very efficient in computing them. Therefore, we have extended the KEP by an external logic block to compute the expressions. For ESTEREL programs with signal expressions, we get a significant reduction of energy consumption and execution time, both for the theoretical computed WCRT and the actually measured execution time. The extra resource usage is small compared to the KEP, but we have to introduce additional input and output signals. Both the improvement and the number of additional signals depend on the actual program.

So far, we only consider signal expressions and expressions on boolean values. More complex expressions, involving arithmetics, would be possible, but would be significantly more hardware intensive.

A limitation of the co-synthesis approach as currently implemented is that we have to generate new hardware for each program. Since the KEP is so far implemented on a FPGA, this is not a real problem yet. However, to also make this co-synthesis approach attractive in case the software part is executed on an ASIC or a standard processor, it might be interesting to employ a "programmable" combinational logic, using off-the shelf FPGAs or PLAs or some custom design. This would lead to new timing problems when the processor runs with a higher clock frequency than the FPGA. But due to the scheduling mechanism, the minimum time to compute a signal expression can be obtained statically. Therefore, the compiler should be able to order the instructions in such a way, that all values are computed in time.

As noted before, the source-level partitioning presented here should be applicable in any software-based Esterel processing whenever it is desirable to delegate (instantaneous) computations to external units. We would therefore would like to explore uses of this source-level partitioning that go beyond reactive processing.

5. REFERENCES

[1] BALARIN, F., GIUSTO, P., JURECSKA, A., PASSERONE, C., SENTOVICH, E. M., TABBARA, B., CHIODO, M., HSIEH, H., LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A., AND SUZUKI, K. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach.* Kluwer Academic Publishers, Apr. 1997.

[2] BENVENISTE, A., CASPI, P., EDWARDS, S. A., HALBWACHS, N., GUERNIC, P. L., AND DE SIMONE, R. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems* (Jan. 2003), vol. 91, pp. 64–83.

[3] BERRY, G. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London 339* (1992), 87–104.

[4] BERRY, G. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999. ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

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

[6] EDWARDS, S. A. CEC: The Columbia Esterel Compiler. http://www1.cs.columbia.edu/~sedwards/cec/.

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

[8] JIANG, Y., AND BRAYTON, R. K. Software synthesis from synchronous specifications using logic simulation techniques. In *DAC '02: Proceedings of the 39th conference on Design automation* (New York, NY, USA, 2002), ACM Press, pp. 319–324.

[9] LI, X., BOLDT, M., AND VON HANXLEDEN, R. 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).

[10] MICHELI, G. D., ERNST, R., AND WOLF, W., Eds. *Readings in Hardware/Software Co-Design.* Morgan Kaufmann, 2001.

[11] PANDYA, P. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. In *Electronic Notes in Theoretical Computer Science* (2002), F. Maraninchi, A. Girault, and Éric Rutten, Eds., vol. 65, Elsevier.

[12] ROOP, P. S., SALCIC, Z., AND DAYARATNE, M. W. S. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)* (Pisa, Italy, Sept. 2004). http://www.ece.auckland.ac.nz/~embsys/publications/emsoft.pdf.

[13] TARDIEU, O. Goto and Concurrency—Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)* (Barcelona, Spain, Mar. 2004).

[14] VON HANXLEDEN, R., LI, X., ROOP, P., SALCIC, Z., AND YOONG, L. H. Reactive processing for reactive systems. *ERCIM News 66* (Oct. 2006), 28–29. http://www.ercim.org/publication/Ercim_News/EN67.pdf.