HW/SW Co-Design for a Reactive Processor

Sascha Gädtke, Xin Li, Marian Boldt,
Reinhard von Hanxleden [1]

Department of Computer Science, Christian-Albrechts-Universität Kiel,
Christian-Albrechts-Platz 4, D-24118 Kiel, Germany

ABSTRACT

This paper presents an approach to accelerate reactive processing via an external logic block that handles complex signal expressions. A reactive program, programmed in the synchronous language Esterel, is synthesized into a software component, running on the Kiel Esterel Processor, 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.

KEYWORDS: Reactive Processing, Hardware/Software Co-Design, Synchronous Languages, Esterel

1 Introduction

The concurrent synchronous language Esterel [Berr92] is typically translated into other, non-synchronous languages such as VHDL for hardware synthesis or C for software synthesis, in the latter case using traditional microprocessors as execution platform. There have also been proposals to combine a classical processor with synthesized hardware, as in the POLIS co-design system [Bala97], which uses the Extended Finite State Machine (EFSM) model of computation. Another simulation-based scheme, also using the EFSM model, employs sophisticated logic minimization for optimization [Jian02].

Another approach that has emerged recently is *reactive processing*, exemplified by the EMPEROR [Daya05] or the Kiel Esterel Processor (KEP) [Li06]. In particular reactive control flow constructs, such as preemption and concurrency, can be processed very efficiently. However, it turns out that the processing of complex Esterel signal expressions is still relatively inefficient compared to hardware. As in the traditional software-based approach, the computation of such expressions must be sequentialized into a series of instructions. As an example, consider the Esterel module EXAMPLE shown in Figure 1(a). This program first tests whether the signal expression (A and C) or (B and C) is present; if so, it emits signal O. Then the program tests whether A or B are present; if so, it emits signal P. The corresponding KEP assembler code is shown in Figure 1(b). The first actual instruction is the PRESENT statement in line 7. This statement tests signal A; if it is present, control moves on to the next statement; if it is absent, control is transferred to label A0 (line 10). With a series of such conditional jumps in lines 7–11 and an interspersed GOTO statement the first signal expression (A and C) or (B and C) is evaluated, and the EMIT O statement is executed accordingly. Similarly, lines 13–15 compute the second signal expression. All told, a total of eight instructions are used to compute an expression for which a hardware realization would use just a couple of logic gates.

This example is certainly an extreme case, as it is dominated by the computation of signal expressions. For such an application a realization completely in hardware would probably be desirable.

[1] E-mail: {sga,xli,mabo,rvh}@informatik.uni-kiel.de

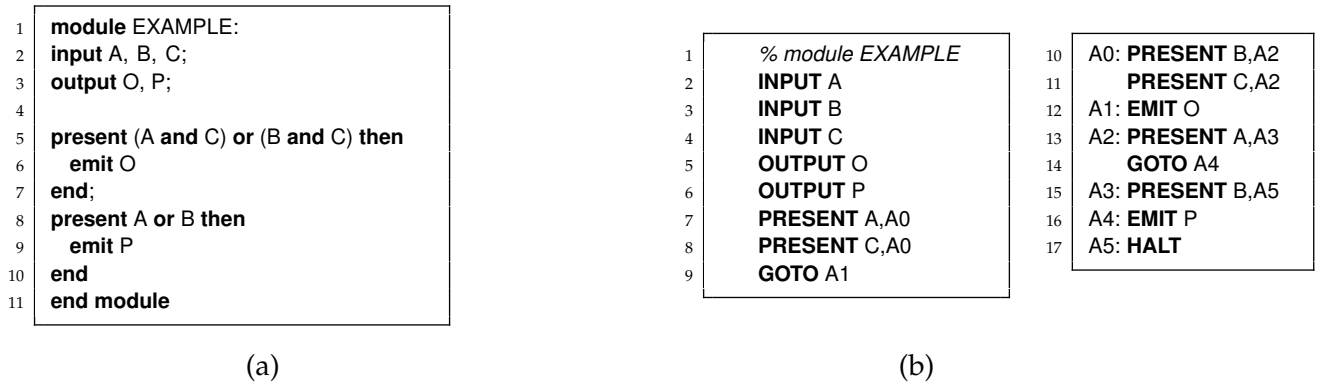


Figure 1: An Esterel example and the resulting KEP assembler code.

However, a more common case is a combination of involved reactive control flow, for which the reactive processing approach is advantageous, and the computation of signal expressions, for which a hardware solution would be desirable. Thus motivated, this paper proposes an approach that extends reactive software processing with an automatically synthesized hardware component that handles the computation of signal expressions.

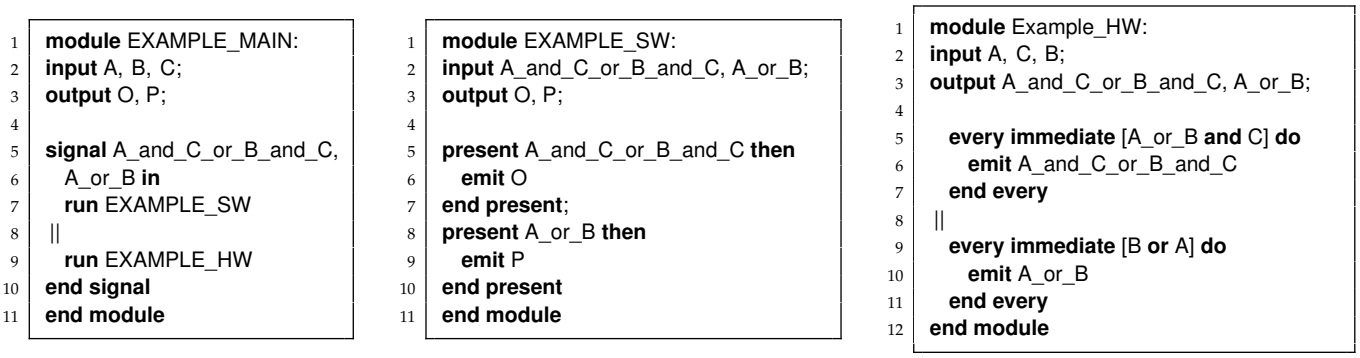
2 Approach

A possible approach to transform an Esterel program into a HW/SW co-design would be to fold the extraction of compound signals into a compiler that would transform an Esterel program directly into reactive processing code plus the hardware logic plus the interface between them. However, this would limit the re-use of existing synthesis tools, and it would make the validation of our partitioning unnecessarily difficult. We have therefore opted for a two-step approach, with an optional intermediate minimization step.

2.1 First Step: Source Code Transformation

In the first step, the source code is transformed into an equivalent Esterel program consisting of a concurrent *software* and *hardware module*, and a *main module* that runs the hardware module and the software module in parallel. First, the Esterel program is analyzed for the presence of compound signal expressions (such as A or B). The aim is to move the computation of such signal expressions from the software module, which will be executed on the reactive processor, onto the hardware module; we therefore will refer to the hardware module also as *logic block*. To make this possible, the reactive processor must make the individual signals that contribute to this expression (such as A and B) available to the logic block. From the perspective of the reactive processor, these signals are treated just like ordinary output signals. Similarly, the logic block makes the result of the compound expressions available to the reactive processor, as another signal (such as A_or_B). From the perspective of the reactive processor, this is treated just like an ordinary input signal.

For the EXAMPLE module, the resulting main module is shown in Figure 2(a). The software module, shown in Figure 2(b), is a copy of the original program with all compound signal expressions replaced by unique auxiliary signals. Every auxiliary signal is introduced as a new input to the software module, as local signal to the main module, and as output signal to the hardware module. The hardware module, shown in Figure 2(c), consists of a series of concurrent threads. Each thread is responsible for the generation of one auxiliary signal *AuxSig* that is present whenever the signal expression *SigExp* is present. To implement this, each thread has the form "every immediate [*SigExp*] do



```
module EXAMPLE_MAIN:
input A, B, C;
output O, P;

signal A_and_C_or_B_and_C,
  A_or_B in
  run EXAMPLE_SW
||
  run EXAMPLE_HW
end signal
end module
```

(a)

```
module EXAMPLE_SW:
input A_and_C_or_B_and_C, A_or_B;
output O, P;

present A_and_C_or_B_and_C then
  emit O
end present;
present A_or_B then
  emit P
end present
end module
```

(b)

```
module Example_HW:
input A, C, B;
output A_and_C_or_B_and_C, A_or_B;

  every immediate [A_or_B and C] do
    emit A_and_C_or_B_and_C
  end every
||
  every immediate [B or A] do
    emit A_or_B
  end every
end module
```

(c)

Figure 2: The EXAMPLE module from Figure 1, after source-level transformation into a HW/SW-partitioned program.

emit *AuxSig* end every".

To minimize the external logic block, we also perform standard logic minimizations. For that purpose we employ the MVSIS package [Jian02], with the fullsimp transformation, which transforms the signal expression into two-level logic, and the fxu optimization, which extracts common factors. In the example, this minimization resulted in the extraction of A or B as common factor.

2.2 Second Step: HW/SW Synthesis

After the first step, we could still synthesize the transformed program into software or hardware. This allows for example to validate the transformation by comparing the behavior of the all-software version of the transformed program to the behavior of the original program. To generate a HW/SW co-design in the second step, the software module is translated to KEP assembler code, and the hardware module is translated into a VHDL description of a logic block.

For the software synthesis, we can employ the KEP Esterel compiler without further modification. The computation sequences for the signal expressions, which were needed in the pure software solution, are now replaced with single PRESENT instructions that test for the presence of the corresponding auxiliary signal generated by the logic block.

For the synthesis of the logic block, we might also use one of the Esterel compilers that can generate VHDL, such as Esterel V5/V7 or the CEC [Edwa]. However, it turns out that these compilers, which were developed to synthesize the full range of Esterel statements, generate too much overhead for the simple logic synthesis task at hand here. We have therefore developed a simple hardware compiler from scratch, which is specifically geared towards the synthesis of the combinational logic hardware modules resulting from our partitioning transformation.

3 Experimental results

To evaluate the effectiveness of the co-design approach, we compare a pure software implementation with a software solution combined with an external logic block. Table 1 shows the results for the TCINT benchmark from the EstBench suite [CEC]. For the original program, the table gives the code sizes and execution times for the Microblaze, generated by the Esterel V5 and V7 compilers and the CEC, and the code size for the KEP. On the Microblaze, the V7 compiler produces the most compact and also the fastest code. The KEP code of the original program is already quite a bit smaller and faster than the fastest Microblaze code.

		Original program (SW)				Partitioned program (SW)				HW+SW
		MicroBlaze			KEP	MicroBlaze			KEP	KEP
		V5	V7	CEC		V5	V7	CEC		
Memory (in bytes)		14860	**11376**	15340	3527	17308	**11416**	17460	4471	1894
Clock cycles	Average	3488	**1797**	2121	261	4248	**1826**	2971	720	204
	Maximum	3580	**1878**	2350	729	4336	**1907**	3311	981	345
	Empty input	3476	**1807**	2101	237	4267	**1838**	2956	771	204

Table 1: Experimental results for the TCINT benchmark.

For the partitioned program, we first again evaluate the performances of a pure software implementation. Again the V7 compiler produces the best code for the Microblaze, but slightly worse than before the transformation. The KEP code has become significantly worse than for the original program, but is still faster than the Microblaze.

Finally, the table gives the code size and performance for the co-design. Compared to the pure software solution on the KEP, the code size has been reduced almost by half. The signal delays within the external logic are negligible and are hidden by other signal delays within the KEP, hence the clock rate is not affected. Therefore, the significant reductions in the cycle counts translate directly into performance gains. The maximum execution time (345 cycles) is reduced to less than half of the pure software solution (729 cycles).

4 Conclusions

This paper has presented a co-design approach to accelerate reactive processing by using external logic blocks. The separation into a source-level transformation, followed by standard software synthesis and customized hardware synthesis, facilitates validation and optimization. One issue, not addressed here for space considerations, is signal reincarnation, which we address with an extended transformation procedure not presented here.

So far, this transformation considers only pure signal expressions. Another issue, not addressed yet, is the handling of the pre operator. We are also investigating extensions to valued signals and variables.

References

[Bala97] F. BALARIN, P. GIUSTO, A. JURECSKA, C. PASSERONE, E. 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, April 1997.

[Berr92] G. BERRY AND G. GONTHIER. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

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

[Daya05] M. DAYARATNE, P. ROOP, AND Z. SALCIC. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, April 2005.

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

[Jian02] Y. JIANG AND R. BRAYTON. Software synthesis from synchronous specifications using logic simulation techniques. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 319–324, New York, NY, USA, 2002. ACM Press.

[Li06] 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.