Reactive Parallel Processing for Synchronous Dataflow

Claus Traulsen and Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24118 Kiel, Germany
{ctr, rvh}@informatik.uni-kiel.de

ABSTRACT

The control flow of common processors does not match the specific needs of reactive systems. Key issues for these systems are preemption and concurrency, combined with timing predictability. To model reactive systems, synchronous programming languages are well-suited, which can be either synthesized to hardware or compiled to C and run on a normal processor. Both of these approaches have significant drawbacks: the generation of hardware is inflexible, the timing analysis of the generated C code is complicated.

We propose a special parallel processor, designed to execute programs written in the synchronous dataflow language Lustre, or its graphical variant Scade. This approach achieves an efficient but still predictable execution. We introduce the processor as well as compiler from Lustre and Scade. To validate our approach, we compare a prototype of the processor, running on an FPGA, with a MicroBlaze processor that executes C code generated from Lustre programs.

Keywords

Reactive processors, parallel execution, synchronous languages, synchronous dataflow, Lustre, Scade

1. INTRODUCTION

Reactive systems are control systems that have to react continuously to inputs at a rate that is determined by their physical environment. The control-flow of such systems differs from that of standard computer systems: the systems are inherently concurrent, since they have to deal with a physical environment that itself is concurrent. The control can often be in different *modes*, hence parts of the systems have to be *suspended* or *aborted*. Since these systems are often highly safety-critical, the behavior of the controller should always be deterministic. This is not only true for the functional behavior, but for the timing as well. To match these needs, synchronous languages, such as Esterel [16], Lustre [9], and Signal [6] were introduced. The execution of these languages is divided into discrete *ticks*. While Esterel is an imperative, control-oriented language, Lustre and Signal are dataflow languages. Since the design of reactive systems is often done by engineers who have a background in control theory, rather than in computer-science, a dataflow formalism is a good means to describe these systems.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Predictable/Reactive Processors

For processor design, the usual approach is to "make the common case fast." However, this is not true for reactive systems, where we must react in time under all circumstances. Here we need the worst case to be fast and we must be able to statically determine the Worst Case Reaction Time (WCRT), or at least a tight upper bound for it. Recently, there have been multiple approaches to build a processor that allows tight timing analysis, such as the PReT [14]. Here, the goal is to design a general purpose processor with timing analysis in mind, while still allowing the execution of arbitrary C code. Another approach are the so called *reactive processors*, which are designed to execute programs written in synchronous languages. The Kiel Esterel Processor (KEP) [12, 13], the EMPEROR [22] and the STARPro [23] are reactive processors that can directly execute Esterel programs. While this limits the class of executable programs, it greatly simplifies both processor design and timing analysis. Reactive processors can be seen as an Application Specific Instruction-set Processor (ASIP), or since they are not designed for a specific application but for the whole application area, namely reactive systems, as an Application Area Specific Instruction-set Processor (AASIP). To analyze the WCRT for the execution on reactive processors is much simpler than for standard processors [3, 15], because they optimize, like synchronous languages, for the worst case.

Developing robust and correct software for reactive systems is not trivial. One of the problems is to ensure that the actual implementation performs the same way as the high-level model, on which formal verification can be performed. While there exist certified compilers for Esterel and Scade, this only ensures that the compilers were developed with a specified, robust methodology, but it does not ensure that they are actually correct, *i. e.*, always produce code that behaves the same way as the model. In particular, when the models are translated into a subset of C, as it is done for Esterel and Scade, it is hard to ensure the

correctness of the compilation. Here reactive processors can help by giving a simple instruction set architecture that is specially designed to support the modeling language. This makes the compilation process easier and should allow for a provably correct compiler. Of course this also implies that the processor itself must be proven to be correct, but this seems to be a simpler task than proving the correctness of software for general-purpose processors. Another benefit is the increased traceability. The compilation of synchronous programs to traditional instruction sets makes it hard to determine which statement in the source program led to which assembler instruction. In contrast, the instruction set of a reactive processor allows for a direct mapping between the assembler and the source model: low-level debugging can be performed on the high-level model.

One of the advantages of synchronous programs is their deterministic behavior, which is independent from the scheduling. This is also true in complex situations, *e. g.*, multiple interrupts, without relying on the precise timing as common processors. Reactive processors enforce determinism in two ways. First, like synchronous languages, they sample their inputs so that all input values are unique in one tick. Second, events occur either simultaneously, or they are separated by at least one tick. Reactive processors avoid race conditions, either by mapping concurrent threads to hardware threads with clearly defined switch points, or by implementing concurrency directly, but enforcing the uniqueness of all values within a tick. To achieve this, the compiler or the processor itself has to ensure that all writes to a specific value are performed before the value is read.

Contributions, Outline

In this paper, we propose a special processor, the Kiel Lustre Processor (KLP) [1], designed for efficient and predictable execution of synchronous data-flow programs. Its key ideas are to use the implicit concurrency in Lustre programs to execute independent equations in parallel and to support Lustre clocks directly, in order to detect which parts of a program need to be executed in one tick. For the scheduling, we evaluate two different approaches, a dynamic scheduling, based on the run-time dependencies, and a priority based scheduling, with statically determined priorities. We also present a compiler from Lustre and Scade, which maps Lustre equations to hardware registers and computes priorities based on the data dependencies. The compiler from Scade also handles automata.

In the next section, we will give an overview on related work. Section 3 introduces the source languages Lustre and Scade. In Section 4, we explain the architecture of the KLP. Section 5 details the compilation process for the KLP. We show experimental results in Section 6, before concluding in Section 7.

2. RELATED WORK

The KEP [12,13] is a reactive processor with an ISA that is closely related to Esterel. It supports concurrency by hardware threads with a priority-based scheduling. Preemption is implemented by hardware watchers, which will sense the code ranges and suspend or abort the execution when their trigger signal is present. In contrast to the KLP, it has only one point of control.

[1]available at `www.informatik.uni-kiel.de/rtsys/kep`

The PReT [14] approach targets the development of a general purpose processor, which can execute arbitrary C code, while still allowing exact timing analysis. To achieve this, processor parts that might have unpredictable timings are replaced by better analyzable parts, *e. g.*, caches by scratch-pad memories and memory access by a memory wheel. However, to fully utilize its features, the programmer needs to use low-level deadline instructions. In contrast, our approach is to use a domain specific input language. This simplifies the processor, and allows to program in a high level programming language.

The Java Optimized Processor (JOP) [18] is designed for time-predictable execution of Java byte-code. It implements a simple pipeline mechanism and cache system that do not introduce timing uncertainties. They use Java byte-code as an input language, hence they have no particular support for reactive control flow.

PRET-C [1] extends C by constructs for deterministic concurrency and preemption and it also defines a small processor extension to perform the scheduling. This approach allows precise timing analysis [17]. In contrast to concurrency in synchronous languages, the scheduling is not based on the data dependencies, but on the syntactical order. Also preemption can only be triggered by values from the previous tick. A similar approach is taken by *SyncCharts in C* [21], an extension of C to directly express synchronous Statecharts. The additional instructions are inspired by the KEP and implemented as C macros. Concurrency is expressed by threads with priorities. To implement preemption, explicit checks for preemption triggers are inserted at all points in the code where control can resume or end in a tick.

Dataflow processors, like the Manchester Machine [7], allow the parallel execution of programs on multiple function units. Here, available data will trigger the execution of instructions that depend on it. This aims for simple parallel execution to reduce the average execution time. Compared to this, our parallelism is more coarse grained, since only data that have reached their final value for the current tick can trigger further executions.

3. LUSTRE AND SCADE

3.1 Lustre

Lustre [8,9] is a synchronous dataflow language. A Lustre program can be seen as a synchronous circuit that not only holds boolean values combined by and/or gates, but also integer values combined by arithmetical operations. Lustre programs are constructed from *nodes*. A node consists of a set of inputs and outputs and a set of concurrent equations, which are executed synchronously.

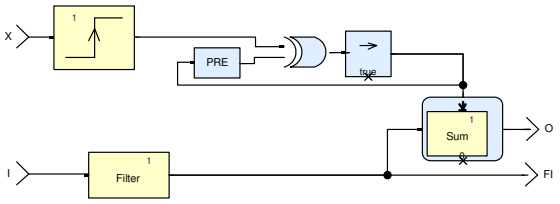
Lustre programs operate on infinite streams of data. *Clocks* define whether a value shall be computed in a tick. In the notion of Lustre, a clock is simply a boolean stream. Lustre defines the following *clock operators*, which allow to access different values in the stream.

e1 → e2 Init: The result is the value of `e1` in the first tick, and the value of `e2` in all following ticks. The first value of `e2` is ignored.

pre e: This gives the value of the expression `e` in the previous tick. In the first tick that `pre` is executed, this value is undefined (nil). In a correct Lustre program, this value

```
1  node ToggledSum(I: int; X: bool) returns (FI, O: int);
2  var on: bool;
3  let FI = Filter(I);
4      on = ToggleOnEdge(true, X);
5      O = current(Sum(FI) when on); tel
6
7  node Filter(In: int) returns (Out: int)
8  let Out = In -> (In+pre(In))/2; tel
9
10 node ToggleOnEdge(init, X: bool) returns (on: bool)
11 var Edge : bool;
12 let Edge = X -> X and not pre(X);
13     on = init -> pre(on) xor Edge; tel
14
15 node Sum(A: int) returns (B:int);
16 let B = 0-> pre(B)+A; tel
```

(a) Lustre implementation



(b) Scade implementation

Figure 1: The ToggledSum example

is never accessed, therefore, pre should only appear on the right hand side of an init statement.

e1 when e2 Down-sample: the expression e1 is only executed when the boolean expression e2 evaluates to true. If e2 evaluates to false, the value is not defined.

current e Up-sample: gives the last defined value of e.

The first node of the Lustre program in Fig. 1a, ToggledSum, takes an integer input I and a boolean X. To the input I it applies a filter, which takes the average of the previous and the current value, and writes the output (FI) to the environment. The input I is also integrated, depending on the value of X. The local signal on is toggled by each rising edge of X, and summation only takes place when on is true. Applying the filter to I and checking whether the integration shall take place at all can be done in parallel, since there are no data dependencies.

A correct Lustre program must not contain cyclic data dependencies. This ensures that the semantically synchronous execution of equations can be translated into a sequential ordering that respects the data dependencies. Furthermore, the program must be *clock consistent*, *i. e.*, when two streams are combined, they must have the same clock.

Lustre programs can be synthesized to hardware or compiled into C code. The original compilation translates the program into finite automata [10]. The current Lustre compiler first checks for clock consistency and implements the clock operators by conditionals. More recent approaches [2] use the clock structure to minimize the generated code.

3.2 Scade

The dataflow equations of Lustre can be naturally represented by dataflow-diagrams. This is done by the language Scade, which is implemented in the SCADE (Safety-Critical Application Development Environment) tool by Esterel Technologies[2]. A Scade version of the ToggledSum example is shown in Fig. 1b, where the content of the Filter and Sum nodes are hidden. To determine the rising edge of the input X, a predefined operator is used. The Sum node is embedded in an activation condition, this corresponds to the current and when in the Lustre program.

Scade extends Lustre by some additional features, such as arrays, convenient conversion between datatypes, and additional build-in functions. It also extends Lustre by Safe State Machines (SSMs), a synchronous Statechart variant [4]. Data equations and automata can be mixed freely, *i. e.*, states may contain equations and the expression, which computes an equation, might itself contain an automaton. Scade supports three different kinds of transitions: *weak-delayed abortions* allow the execution of the source state in the tick the transition is triggered and activate the target state in the next tick. *Strong abortions* immediately abort the execution of the source state when they are triggered and transfer control to the target state. *Synchronized transitions* are triggered when the source state itself reaches a state that is flagged as *final*. The transition triggers can be arbitrary data expressions or signals, *i. e.*, boolean variables.

Since the arbitrary mixing of clocks can lead to complex code that is hard to understand, the usage of clocks in Scade is per default restricted. Clocks are replaced by *activation conditions*, which execute a given node only when a boolean value is true, as the Sum node in Fig. 1b. Whenever the boolean condition is not true, the output is not undefined, as in Lustre, but a default value is used.

4. ARCHITECTURE

The architecture of the KLP (Fig. 2) is designed to directly express the dataflow nature of Lustre programs. Since in Lustre computations are only ordered by their data dependencies, Lustre programs tend to have a high degree of concurrency. The KLP uses this logical concurrency to achieve parallel execution, using two mechanisms. 1) The KLP contains multiple processing units, so that independent computations can be performed truly in parallel. The mapping of instructions to the processing units is done by a hardware scheduler, which either analyses the dependencies at run-time or uses statically set priorities, which are computed by the compiler. 2) The KLP directly implements Lustre clocks, computations that shall not be executed in the current tick are marked as finished without any overhead. In contrast to a *multi-core* approach, all processing units access the same data; the scheduling ensures that no data conflicts occur.

The external interface consists of the following inputs: a Tick signal which triggers the execution of a global tick, a set of integer inputs I, and instructions coming from the instruction ROM. The outputs are the Done signal, to indicate that a tick has finished, a set of integer outputs O, and program counter values PC to request instructions from the ROM. The Exec output is used to generate execution traces.

4.1 Building blocks

The KLP consists of three building blocks. The *register*

[2] www.esterel-technologies.com

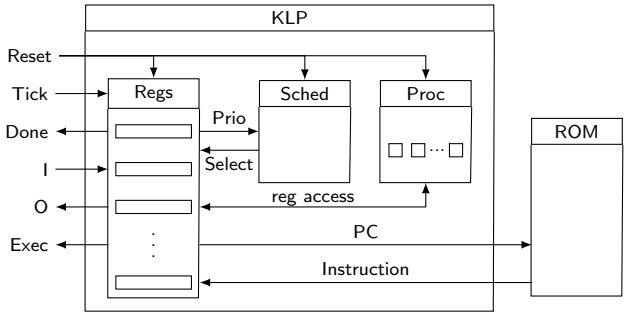


Figure 2: Overview of the KLP

file (Regs) stores all runtime information. For each equation this is: the current value, the previous value, the address of the next instructions to compute this register, the register id, which holds the clock of the expression, and a Done-flag to indicate whether the execution has terminated for the current tick. Additionally, each register has some basic control logic to detect whether it is done, because the register that holds its clock is done and its value is zero. The register file also contains some control logic: whenever the input Tick is set to true, it reads all inputs, overwrites the previous value with the current value for all registers, and resets the Done-flags of all active registers, *i. e.*, registers with a program counter different to zero. Whenever a register has a Done-flag set to false, it sends this information to the scheduler. From the scheduler it receives the information which registers shall be executed. Now it reads the instruction for this register from the instruction ROM, sends the instruction to the processing unit, and waits for the results. When all registers are done, it signals this to the environment via the Done output and writes the output values. When the TICK input is set for the first time, the boot process is started by setting register 0 to ready. This triggers the execution of instruction 0, where the code to initialize the other registers is expected.

The *processing units* (Proc) are responsible for executing the instruction. Each processing unit consists of a dispatcher for the opcodes and an ALU. It gets an instruction and a register id from the register file. It queries the register file for additional register values, if this is necessary to execute the instruction, and writes back the result: a new value, program counter or clock, or the signal to set the Done-flag.

The *scheduler* (Sched) maps ready registers at each tick to free processing units, based on the Done-flags and the priorities of the registers. So far, the scheduler takes the first available registers. Therefore, the actual scheduling and also the utilization are affected by the ordering of the equations: the compiler should order the equations according to the number of dependencies.

To decide which registers are ready and hence which parts of the program can be executed in parallel is crucial for the efficient execution of Lustre programs on the KLP. The KLP implements two alternative ways to achieve this: dynamic scheduling and priority based scheduling.

Dynamic Scheduling

For the dynamic scheduling, each register detects itself whether it is ready to execute, based on the next instruction and the Done-flag of all other registers. It determines the data dependencies for its next instruction and looks up the Done-flags of its arguments. This is similar to traditional out-of-order execution, with all active registers in the event queue, but due to the fixed steps in Lustre, solving data dependencies is much simpler. For each register we know precisely whether the computation of its value has finished for the current tick. On the other hand, this will always check all ready registers, as the instructions in a Lustre are not ordered; therefore it will utilize the maximal degree of parallelism in each instruction cycle. Since we require the dependency graph to be acyclic, at least one register is ready whenever there exists a register that is not done.

Instruction	Remark
SETCLK reg, $clock$	set clock for a register
SETPC reg, pc	set program counter for a register
INPUT id, reg	map register to input id
OUTPUT id, reg	map register to output id
LOCAL reg	mark register as local
DONE pc	set done flag and pc for next tick
PRIO reg, p	set priority for register reg to p
ADD reg_T, reg_A, reg_B	$reg_T \leftarrow reg_A + reg_B$
...	(Integer arithmetic)
AND reg_T, reg_A, reg_B	$reg_T \leftarrow reg_A \& reg_B$
...	(Boolean operators)
LT reg_T, reg_A, reg_B	$reg_T \leftarrow Reg_A < reg_B$
...	(Comparisons)
JMP pc	unconditional jump
JT pc, reg	jump when true
...	(Jumps)
RRMOV reg_T, reg_S	reg to reg move: $reg_T \leftarrow reg_S$
IRMOV reg_T, val	immediate reg move: $reg_T \leftarrow val$

Table 1: Overview of KLP instructions. Registers (reg), immediate values (val), clocks ($clock$), IO identifier (id) and priorities (p) are 1 Byte long, program counter (pc) 2 Bytes.

Priority Based Scheduling

For the priority based scheduling, each register has an additional priority. Two registers may be executed in parallel when they have the same priority. The priorities are computed by the compiler, based on the syntactic dependencies. While this approach may miss potential parallel execution, it needs less control hardware. Another benefit is that it allows non-local dependencies, *i. e.*, dependencies that are not completely determined by the next instruction. Such dependencies are introduced by Scade automata (see Section 5.2), where each outgoing transition adds a dependency to each computation inside a state.

In the example from Fig. 1a, the compiler assigns the highest priority to the equations that compute FI and Edge, the next priority are assigned to on, while O gets the lowest priority. This is reflected in the generated KLP assembler in Fig. 4 by setting priorities 1, 2, and 3, respectively. Note that the lowest number indicates the highest priority.

4.2 Instruction Set

Beside the usual arithmetical and logical instructions, the KLP contains SETCLK and SETPC instructions to initialize a register by setting the clock and the program counter, respectively. The INPUT and OUTPUT instructions map the register to global inputs and outputs.

The DONE instruction marks the current register as fin-

```
1  node ToggledSum (I: int; X: bool;) returns (FI, O: int ;);
2  var on, Edge: bool;
3  let FI = I->((I + pre(I)) / 2);
4      Edge = X->(X and (not pre(X)));
5      on = true->(pre(on) xor Edge);
6      O = current((0->(pre(O) + FI)) when on); tel
```

Figure 3: Clocked equations for ToggledSum

ished for the current tick and sets the program counter for the next tick. It is similar to the pause in SyncCharts in C [21] and the gotopause instruction in Esterel+GOTO [20]. An overview of the instructions is shown in Table 1.

The instruction set of the KLP is regular: each instruction has 32 bits, where the first byte contains the opcode. Furthermore, the first 4 bits of the opcode encode the data dependencies of the instruction, and the second 4 bits encode the ALU-function, in case the instruction uses the ALU.

5. COMPILATION

5.1 Compiling Lustre

The compilation from Lustre into KLP assembler consists of three steps. First, we use the lus2ec tool from the Verimag Lustre compiler[3] to expand all nodes and tuples and to propagate pre operators to variables. This also checks that the programs are well-formed, *i. e.*, every variable is defined exactly once, the dependency graph contains no cycles, and only variables that run on the same clock are combined.

In the second step, we simplify the Lustre program further by restricting the use of clock operators. This results in *clocked equations*, which are mapped to the KLP instruction set in the third step. The first two steps are source to source transformations that yield valid Lustre code. This allows easy validation of the correctness by using existing tools to compare the source file with the generated code.

Clocked equations are Lustre programs where clock operators may only occur at some special positions. We do not allow nesting of pre operators, therefore we might have to introduce additional variables. All equations have the form: x=current((i → e) when C), where i and e are arbitrary expressions that do not contain any clock operators except for pre. The C is either the name of a boolean variable, or true, *e. g.*, the expression is running on the base clock. The Lustre example from Fig. 1a can be expressed by clocked equations as shown in Fig. 3. Note that current ((i → e) when true) can be simplified to i → e. The behavior of a clocked equation directly corresponds to the execution of the KLP: in each tick, we overwrite all previous values. This corresponds to the fact that all equations run on the base clock, *i. e.*, we add a current operator to all equations that have a clock.

Translating clocked equations to KLP assembler is straightforward. For the equation O in the Lustre program (Line 6 in Fig. 3), the following instructions in the KLP assembler (Fig. 4) are generated: in Line 12–15, the register is initialized, by setting its program counter, clock and priority. This register is the only one where a SETCLK instruction occurs, because in the Lustre program O is the only equation that contains a when statement. In Line 15 its priority is set to 3, this is the lowest priority, because O depends on all other processes. Line 14 sets the program counter to Line 32, which initializes the flow with value zero in the first tick, before the program counter the next tick is set to Line 34 by the DONE instruction in Line 33. The actual computation O+FI is performed in Line 34. This code is repeated in every tick by the DONE instruction in Line 35. Because the computation of FI and Edge can be performed in parallel, both registers get the same priority (1) in Line 5 and 8, respectively.

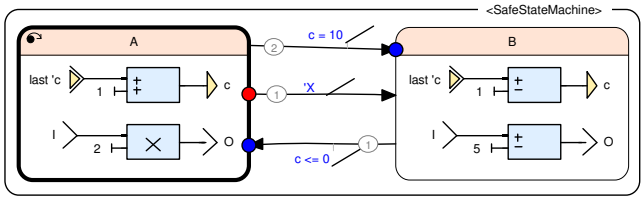
[3] www-verimag.imag.fr/SYNCHRONE/

```
1                INPUT I            // declare input I
2                INPUT X
3                OUTPUT FI          // declare output FI
4                SETPC FI L_FI      // code for FI starts in Line 18
5                PRIO FI 1          // FI has highest priority (1)
6                LOCAL Edge         // declare Edge as local register
7                SETPC Edge L_Edge
8                PRIO Edge 1
9                LOCAL on
10               SETPC on L_on
11               PRIO on 2
12               OUTPUT O           // declare output O
13               SETPC O L_O        // code for O starts in Line 32
14               SETCLK O on        // set clock of O to on (when on)
15               PRIO O 3           // O has lowest priority (3)
16               DONE 0             // setup complete
17
18 L_FI:         RRMOV FI I         // FI = I ->
19               DONE L_FI_run
20 L_FI_run:     ADD FI I pre(I)    // I + pre(I)
21               IDIV FI FI 2       // /2
22               DONE L_FI_run
23 L_Edge:       RRMOV Edge X       // Edge = X->
24               DONE L_Edge_run
25 L_Edge_run:   IXOR Edge pre(X) 1 // not pre(X)
26               AND Edge X Edge    // and X
27               DONE L_Edge_run
28 L_on:         IRMOV on 1         // on = 1 ->
29               DONE L_on_run
30 L_on_run:     XOR on pre(on) Edge // pre(on) xor Edge
31               DONE L_on_run
32 L_O:          IRMOV O 0          // O = 0 ->
33               DONE L_O_run
34 L_O_run:      ADD O pre(O) FI    // pre(O) + FI
35               DONE L_O_run
```

Figure 4: KLP assembler for ToggledSum

5.2 Compiling Scade

As mentioned before, Scade can be compiled into dataflow equations similar to Lustre, which then can be mapped to clocked equations. We propose a direct compilation. Three different approaches exist for the parallel execution of hierarchical automata: the first is to replicate necessary control as it is done by the distributed execution of Lustre programs [5]. Of course, this implies that the same code is executed multiple times. Another possibility is to insert instructions into each parallel branch that explicitly request information of the global state from some master branch, which is for example done by the Emperor [22]. The third possibility is to let the master branch execute the control parts, and only distribute the data-parts that can be easily parallelized. This is the approach we take. For each automaton or macrostate we create one control thread, which runs



(a) A simple Scade automaton

```
1          INPUT I
2          INPUT X
3          LOCAL CTRL
4          SETPC CTRL L_CTRL_A
5          PRIO CTRL 1
6          OUTPUT C
7          SETPC C L_C_A
8          PRIO C 2
9          OUTPUT O
10         SETPC O L_O_A
11         PRIO O 2
12
13 // Controller for state A
14 L_CTRL_A: JT X A_2_B_S  // Check strong abort
15 L_CTRL_AW: PRIO CTRL 3 // Execute state
16           PRIO CTRL 1
17           IEQ T C 10     // Check weak abort
18           JT T A_2_B_W
19           DONE L_CTRL_A // Resume in next tick
20 // Strong abort from A to B
21 A_2_B_S:  SETPC C L_C_B
22           SETPC O L_O_B
23           GOTO L_CTRL_B // Check weak abortion in B
24 // Weak abort from A to B
25 A_2_B_W:  SETPC C L_C_B
26           SETPC O L_O_B
27           DONE L_CTRL_B // Resume in B in the next tick
28 // Controller for state B
29 L_CTRL_B: PRIO CTRL 3
30           PRIO CTRL 1
31           ILE T C 0
32           JT T B_2_A
33           DONE L_CTRL_B
34 // Weak abort from B to A
35 B_2_A:    SETPC C L_C_A
36           SETPC O L_O_A
37           DONE L_CTRL_A // Resume in A in the next tick
38 // Equations inside A
39 L_C_A:    IADD C pre(C) 1
40           DONE L_C_A
41 L_O_A:    IMUL O I 2
42           DONE L_O_A
43 // Equations inside B
44 L_C_B:    ISUB C pre(C) 1
45           DONE L_C_B
46 L_O_B:    ISUB O I 5
47           DONE L_O_B
```

(b) Derived KLP assembler

Figure 5: Handling of Scade automata on the KLP

per default with higher priority than the substates. At each tick it first checks for strong abortions, then lowers its own priority to let the inner equations execute before it raises its priority again, to ensure that strong abortions are checked with high priority in the next thread, and checks for weak abortions. If a transition is triggered, it executes code to reconfigure all equations that are defined in the source and target state. Due to the semantics of SSMs in Scade, each state is executed at most once in each tick, and the only case when a state is entered and left in the same tick is if it is activated by a strong abortion and left by a weak abortion. If a weak abortion is taken, we execute a DONE statement to stop the controller for this tick. For the strong abortion, we still have to check the weak abortions of the target state.

Fig. 5a shows a simple automaton in Scade. It takes two inputs: an integer I and a boolean X. Its only output is the integer O. The local integer variable c is initialized to 0. The variable c is incremented each tick in the initial state A and decremented in state B. Control is transferred from A to B when c equals 10, and from B to A when c is less or equal to 0. These transitions are weak-delayed, indicated by the dot at the arrowhead. Therefore, the equation inside the state is executed one last time in the tick where the transition is triggered, and the execution of the target state starts in the next tick. Control can also be transferred from A to B by the input X. This triggers a strong abortion (dot at the arrow-tail), that will immediately transfer control to state B without executing state A first.

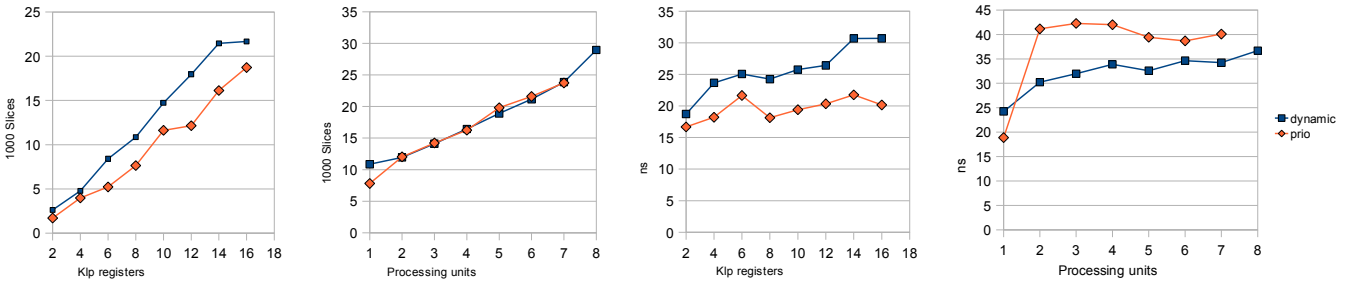
The KLP-assembler for this program (Fig. 5b) consists of the following parts. We assign one register to the control of the complete automaton. (For more complex programs, we need one automaton per hierarchy.) This control part runs with higher priority than the contained equations. In Line 8 the code to check for the execution of state A starts. First, we check whether the trigger X of the strong abortion is true, in this case, we jump directly to A_2_B, which sets the program counter of c and O according to state B. Otherwise, we set the priority to 3, which indicates lower priority than the equations for c and O, which are now executed. Then we raise the priority of the controller back to 1. Thereafter, we check for the weak abortions by comparing C to 0.

The translation from Scade automata to KLP assembler is similar to the translation of SSMs to KEP assembler [19]. The main difference is that the KEP has a single point of control. Therefore, a solution with watchers, which monitor the unique program counter and reset it when a transition occurs, is feasible. There a state is implicitly declared active, when the program-counter is currently inside the scope of the state, while in our approach a state is active when the program counter of the controller is in the corresponding handler.

6. EXPERIMENTAL RESULTS

The KLP is developed in Esterel v7 with Esterel-Studio, from which a software emulation in C and a hardware description in VHDL is generated. For evaluation purposes, we extend it by a test-driver that can communicate via a simple protocol to set inputs, read outputs, load programs, and get information on the current execution, such as the execution trace or the reaction time. The test-driver communicates either via the serial port, when run on an FPGA, or via TCP/IP for the software emulation. To validate our approach, we compare the generated outputs to the results of the Lustre v4 compiler. We use the tool lurette [11] to generate random traces for our benchmarks, which are then executed on the KLP. The generated outputs are compared to the outputs, when the traces are run on the benchmarks compiled with the Lustre v4 compiler.

Fig. 6 shows the number of slices and the minimal instruction cycle when it is synthesized for a Virtex 4 FPGA, depending on the number of registers and processing units for the dynamic scheduling (dynamic) and the priority based



(a) Slices per register, for 1 processing unit

(b) Slices per processing unit, for 8 registers

(c) Minimal delay per register, for 1 processing unit

(d) Minimal delay per processing unit, for 8 registers

Figure 6: Hardware usage of the KLP

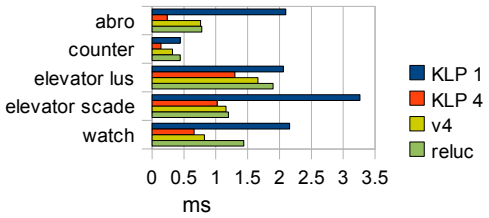


Figure 7: Reaction times on the KLP compared to the execution of the program on a MicroBlaze soft-core, using different Lustre compilers. The KLP uses 1 or 4 processing units, respectively.

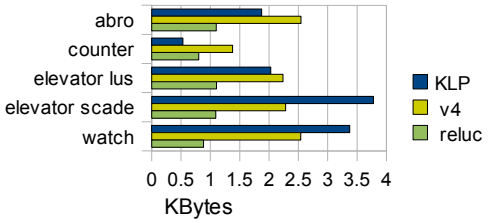


Figure 8: Code size, compiled for the KLP or for standard hardware using different Lustre compilers

scheduling (prio). The scalability per register of the priority based approach is slightly better, both with regard to the usage of slices and the minimal delay. When adding more processing units, the usage of slices is about equal for both approaches. The priority based approach is slower in this case, because the scheduling is more involved. This also explains why the minimal delay is doubled when going from one processing unit to two. As to be expected, adding additional processing units scales better than adding registers.

We measure the reaction times for a set of benchmarks when running on a Microblaze core with 100 MHz and compared it to the runtime on the KLP, with 1 or 4 processing units. We synthesized the KLP to get the maximal possible frequency and used the software emulation to measure the number of instruction cycles that are needed to compute one reaction. Fig. 7 shows the measured worst case reaction times. The KLP performs better than the code compiled by the Lustre v4 compiler. It performs about equal to the code generated by the commercial reluc compiler. Our compiler performs a basic, syntax-based mapping with minor optimizations, while the reluc compiler performs more aggressive optimizations. For these benchmarks, the 4 processing units cannot always be used. Since the scheduling for the 4 processing units enforces a lower frequency, the computation of small benchmarks with few equations or for benchmarks with many data dependencies can even be slower for more processing units.

Fig. 8 compares the generated code size for the KLP with code generated by different Lustre compilers: the Lustre v4 compiler from VERIMAG and the reluc compiler from SCADE.

For the KLP-compiler we measure the code size of the KLP-assembler. For the other compilers, we generate C code and compile it further with the gcc into object code for a standard PC. The code of the KLP is relatively large. This has two, first we need additional code due to the parallel execution: for each register we need a DONE instruction both for the runtime code and the initialization plus the setup code. To reduce this overhead, the compiler could analyze dependencies and combine registers that cannot be executed in parallel. And the compiler only performs limited optimizations, in particular compared to the reluc compiler. Note that for the simple counter example, which has few possibilities for optimization, the code for KLP is smaller than for the other compilers.

7. OUTLOOK

We have presented the KLP, a processor which is specially designed to execute synchronous dataflow programs in parallel. It directly supports clocks to determine which code must be executed in one tick, and it supports the synchronous execution of parallel equations. We also showed the compilation process from Lustre and Scade into the KLP-assembler. Experiments show that the execution times are competitive with the compilation from Scade. However, the efficiency of the processor description needs to be improved. The simple architecture of the KLP allows easy timing analysis.

One problem in modern processor design is the latency of memory access, processors can execute instructions faster than they can be transferred from the instruction ROM. We do not address this problem here. One possible way to cope with this problem is to introduce an instruction cache. To simplify the performance estimation, the cache should be aware of the registers, hence it should catch instructions for each register independently. To reduce hardware usage,

we could distinguish between valued registers and boolean registers, which hold clocks. This can be done easily, since the register kind is fixed for each instruction.

Also for the compilation are many optimizations possible. For example, to force initialization, clocks often have the form $\mathsf{true} \rightarrow \mathsf{C}$ where C is some input. So far, we copy the input in each tick but the first. Instead, we could simply change the clock-register after the initialization. So far, the compiler requires one register for each flow in the Lustre program. If a program needs more registers than available in the KLP, the compiler could implement these by combining clocks and implement clocks by conditionals, as it is done by the compilation from Lustre to common processors.

8. REFERENCES

[1] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, INRIA Grenoble Rhône-Alpes, 2009. `http://hal.inria.fr/docs/00/39/16/21/PDF/rr.pdf`.

[2] D. Biernacki, J.-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, AZ, USA, June 2008.

[3] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008. Proceedings of the International Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'07), March 2007, Braga, Portugal.

[4] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey City, NJ, USA, Sept. 2005.

[5] A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *3rd International Conference on Embedded Software, EMSOFT '03*, volume 2855 of *Lecture Notes in Computer Science (LNCS)*, pages 206–222, Philadelphia, PA, USA, Oct. 2003. Springer-Verlag.

[6] P. L. Guernic, T. Goutier, M. L. Borgne, and C. L. Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), Sept. 1991.

[7] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 1985.

[8] N. Halbwachs. A synchronous language at work: the story of Lustre. In *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'2005*, Verona, Italy, July 2005.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[10] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13207–218. Springer Verlag, 1991.

[11] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer*, 8(6), Nov. 2006.

[12] 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.

[13] 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.

[14] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*, Atlanta, USA, Oct. 2008.

[15] M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, Apr. 2009.

[16] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.

[17] 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.

[18] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2), 2008.

[19] F. Starke, C. Traulsen, and R. von Hanxleden. Executing Safe State Machines on a reactive processor. Technical Report 0907, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, Mar. 2009.

[20] O. Tardieu. Goto and Concurrency—Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'04)*, Barcelona, Spain, Mar. 2004.

[21] R. von Hanxleden. SyncCharts in C. In *Proceedings of the International Conference on Embedded Sofware (EMSOFT'09)*, Grenoble, France, Oct. 2009.

[22] 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.

[23] 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.