An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis

Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Dept. of Computer Science and Applied Mathematics
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{xli,jlu,mabo,miha,rvh}@informatik.uni-kiel.de

ABSTRACT

The concurrent synchronous language Esterel allows programmers to treat reactive systems in an abstract, concise manner. An Esterel program is typically first translated into other, non-synchronous high-level languages, such as VHDL or C, and then compiled further into hardware or software. Another approach that has been proposed recently is the *direct execution* of Esterel-like instructions with a customized processor, which promises the flexibility of a software solution with an efficiency close to a hardware implementation. However, the instruction sets and implementations of the processor architectures proposed so far still have some limitations regarding their completeness, efficiency, and adherence to the original Esterel semantics. This paper presents a novel reactive processor architecture, the Kiel Esterel Processor, which addresses these shortcomings. In particular, it provides a complete, semantically accurate implementation of the Esterel preemption primitives, most of which can be expressed directly with a single machine instruction.

One advantage of the reactive processors—in addition to their high execution speed compared to traditional software implementations—is that control-flow is preserved while compiling Esterel into machine code, and that the execution platform has a very predictable timing behavior. This paper presents a precise and very efficient *Worst Case Reaction Time* (WCRT) analysis, which is geared towards the Kiel Esterel Processor, but which could be adapted to other reactive processors as well.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Other Architecture Styles—*High-level language architectures*; D.3.4 [**Programming Languages**]: Processors—*Code generation*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

CASES'05, September 24–27, 2005, San Francisco, California, USA.

General Terms

Languages, Performance

Keywords

Synchronous languages, Esterel, reactive processing, WCET, reaction time analysis

1. INTRODUCTION

Many embedded systems belong to the class of *reactive systems*, which continuously react to inputs from the environment by generating corresponding outputs. The programming of reactive systems typically requires the use of non-standard control flow constructs, such as concurrency or exception handling. Most programming languages, including languages such as C and Java that are commonly used in embedded systems, either do not support these constructs at all, or their use induces non-deterministic program behavior, regarding both functionality and timing. To address this difficulty, the synchronous language Esterel [5, 4] has been developed to express the control flow patterns typically found in reactive systems in a concise manner, with a clear semantics that imposes deterministic program behavior under all circumstances.

Esterel programs are mostly implemented in one of two ways. In the software synthesis approach [5, 10], the Esterel program is translated by an Esterel compiler into a C program, which in turn runs on a COTS processor. Esterel can also be synthesized into hardware directly [2], via some hardware description language such as VHDL. For a hybrid approach, Esterel can also be used in hw/sw co-design [1]. Recently, another alternative has emerged, where the Esterel program is running on a processor that has been developed specifically for the purpose of executing Esterel. The instruction set of these *reactive processors* closely resembles the constructs found in Esterel, such as waiting for occurrence of a signal or abortion. We distinguish two variants of this approach. The *patched reactive processor* implementation combines a COTS processor core with an external hardware block that implements additional Esterel-style instructions. A *custom reactive processor* implementation consists of a full-custom *reactive core*, whose instruction set and data path have been tailored exclusively for the processing of Esterel code. So far, there have been only limited and fairly recent investigations of the reactive processor approach. To our knowledge, the ReFLIX and RePIC architectures proposed by Dayaratne, Roop, Salcic *et al.* [17, 18, 9] are the

only ones that fall into this category, and they both follow the patched processor strategy. Their results are fairly promising, illustrating the potential of this approach. However, there are also certain limitations of the architectures proposed so far, for example regarding their support of the Esterel preemption primitives; see also Section 2.

In this paper, we present an alternative architecture, the *Kiel Esterel Processor* (*KEP*), which is a custom reactive processor, to our knowledge the first of this kind. The architecture presented in this paper is version 2.0 of the Kiel Esterel Processor, hence we also refer to it as KEP2. Notable features of the KEP2 include the following:

1. It gives a complete, semantically accurate implementation of the Esterel preemption primitives, including weak and strong abortion and suspension.

2. As the instruction set and data path have been developed specifically for Esterel execution, the individual machine instructions can be executed fairly fast. Furthermore, most typical Esterel commands can be expressed directly with just a single KEP command, improving speed further and leading to minimal instruction and data memory usage.

3. The KEP also includes an interface block for handling input and output signals, which directly supports testing the presence and values of signals across logical instants (corresponding to Esterel's pre operator).

4. A *tick manager* ensures that logical ticks are executed at a pre-defined frequency, and indicates timing violations at run time.

5. Throughout the development of the KEP, scalability has been a consideration, hence the allowed number of signals, the nesting depth of preemption primitives, and other design parameters are fully configurable.

Unlike traditional processors, the reactive processors offer instructions to process most of the reactive constructs directly. Therefore, the reactive processing approach tends to be significantly more efficient than the traditional software synthesis approach. The reactive processing approach is particularly suited to implement reactive systems; we envision that the reactive processing approach is suitable for applications where a custom hardware design is not appropriate, for example due to low volume, short development cycles or the need for configurability or customization, and where a software design would be too slow or too resource consuming. However, as we wish to demonstrate in this paper, the reactive processing approach also lends itself very well to implementing and analyzing systems whose timing behavior is critical. Such *hard real-time systems* are characterized by the need to meet certain given deadlines when performing their computations. Hard real-time systems are often safety-critical; typical examples include airbag controllers, flight-control software, or medical applications. A traditional difficulty encountered when developing software for hard real-time applications is that the execution time of this software is difficult to predict. (In fact, in the general case it is impossible for Turing-complete languages.) To perform a *Worst Case Execution Time* (WCET) analysis on a piece of code typically imposes fairly strong restrictions on the analyzed code, such as a-priori known upper bounds on loop iteration counts, and even then control flow analysis is often overly conservative [15, 6]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict exactly how much time the execution of these instructions consumes, due to pipelining, out-of-order execution, argument-dependent execution times (*e. g.*, particularly fast multiply-by-zero), and caching of instructions and/or data. Finally, if external interrupts are possible or if an operating system is used, it becomes even more difficult to predict how long it really takes for an embedded system to react to its environment. To summarize, performing conservative, yet tight WCET analysis appears by no means trivial and is still an active body of research. In fact, despite the advances already made in the field of WCET analysis, it appears that most practitioners today still resort to extensive testing plus adding a safety margin to validate timing characteristics. However, as we want to demonstrate here using the case of the KEP, timing analysis is practical for reactive processors based on Esterel, hence making the reactive processing approach particularly well suited for hard real-time systems. As we here are investigating the timing behavior for reactive systems, we are concerned with computing the maximal time it takes to compute a single reaction, that is the time from given input events to generated output events. Hence, we call this analysis a *Worst Case Reaction Time* (WCRT) analysis.

After a discussion of related work in the following section, Section 3 gives an introduction into Esterel's synchronous execution model and into the KEP, also outlining the translation scheme we are using to create KEP assembler from Esterel. Section 4 presents a Worst Case Reaction Time Analysis scheme for the KEP, followed by experimental results in Section 5 and conclusions in Section 6.

2. RELATED WORK

As mentioned in the introduction, the only other reactive processor proposals in the sense of Esterel that we are aware of are the ReFLIX and RePIC designs [17, 18, 9], of which RePIC is the more advanced. The RePIC includes an abort handling block, which is used for handling both strong and weak aborts; it does not handle suspension. These abort types are distinguished by the judicious placement of additional instructions in the assembler code. For every layer of enclosing abort, a chkabort instruction must be executed at every instant; for comparison, our approach does not require such additional instructions, as the abort handling is performed simultaneous to executing the abort block itself.

Figure 1(a)/(b) shows an example from Dayaratne *et al.* [9] for translating the Esterel weak abort statement to the RePIC assembler. The abort handler is configured with a pair of instructions; ldaaddr (line 4) specifies the continuation address, and abort (line 5) indicates the trigger signal. For weak abortion, the RePIC inserts a chkabort instruction before every await instruction within the abort body to determine whether control stays within the abort body in that logical instant. For example, after emitting signal Z (line 8), the chkabort (line 9) determines whether the "await B", encoded in lines 10–12, should be executed next, or whether a jump to the continuation address should be performed, thus aborting the body. As presented there, this would respond correctly to the presence of the abort signal A; however, if A is absent, execution would reach the await-loop in lines 10–12, and would in the following ticks only be sensitive to

```
% Esterel
...
weak abort
  ...
  emit Z;
  await B;
  emit X;
  await C;
  emit W;
when A;
emit Y;
...
```

(a)

```
% RePIC
  ...
4  ldaaddr19
5  abort 0 A
   ...
8  emit Z
9  chkabort 0
10 await
11 present B
12 goto 10
13 emit X
14 chkabort 0
15 await
16 present C
17 goto 15
18 emit W
19 emit Y
   ...
```

(b)

```
% KEP
...
WABORT 1,A,A0
...
EMIT Z
AWAIT B
EMIT X
AWAIT C
EMIT W
A0: EMIT Y
...
```

(c)

Figure 1: An Esterel code fragment involving a weak abort statement (a), and the corresponding assembler code for RePIC (b) and KEP (c).

```
% Esterel
module ABRT:

input A;
output S, T;

abort
  emit S;
  halt;
  emit T
when A
end module
```

(a)

```
% KEP Assembler
% module ABRT
INPUT A;
OUTPUT S, T;

EMIT _TICKLEN, #6
                    % T0
ABORT 1, A, A0      % N1.2
  EMIT S            % N1.3
  HALT              % N1.4, T2
  EMIT T
A0:                 % N3.0
HALT                % N3.1, T4
```

(b)

Figure 2: Illustration of abort/halt in Esterel (a) and the corresponding KEP assembler (b).

the awaited signal B, not to the abort trigger signal A. It seems that this could be remedied by including the chkabort instruction within the await-loop (*i. e.*, changing the "goto 10" in line 12 to "goto 9"). However, what we see as the more significant limitation is that this abort handling mechanism seems not as efficient as it could be, especially when considering nests of aborts. For comparison, the KEP handles aborts directly in hardware, without the need for additional assembler instructions to check the presence of abort signals at each control point, thus resulting in more compact and efficient code. To illustrate, consider the KEP assembler shown in Figure 1(c) for the same Esterel example; a single WABORT assembler instruction configures the abort handler, which is then active concurrently with the execution of the abort body, and which autonomously performs the necessary preemption of the abort body (correctly distinguishing between weak and strong aborts) when the abort signal occurs.

Based on RePIC, Dayaratne *et al.* [9] propose an extension to a multi-processor architecture, called EMPEROR, which allows the handling of Esterel's concurrency operator. This is an interesting approach, which could also be applied to the KEP2 to extend the range of acceptable Esterel programs.

Finally, the KEP2 itself has evolved from earlier designs, the first of which being KEP version 0.1 [12]. This version already implemented the preemption primitives correctly, but did not include the interface block with the support of the pre-operator, did not support variables, did not allow local signals, and did not include the Tick Manager.

As mentioned in the introduction, there exist numerous approaches to classical WCET analysis. Regarding the analysis of synchronous programs, Logothetis, Schneider and Metzler [13, 14] have employed model checking to perform a precise WCET analysis for the synchronous language Quartz, which is similar to Esterel. However, their problem formulation was different from the WCRT analysis problem we were addressing. They were interested in computing the number of ticks required to perform a certain computation, such as a primality test (which we would actually consider to be a transformational system rather than a reactive system); we consider here instead how long it may take to compute a single tick, which can be considered an orthogonal issue.

3. ESTEREL AND THE KIEL ESTEREL PROCESSOR

The execution of an Esterel program is divided into (*logical*) *instants*, or (*logical*) *ticks*, which are conceptually executed infinitely fast. An Esterel program interacts with its environment through *signals*, which are either *present* throughout a logical instant or *absent*. Input signals are sampled at each tick, and each tick may generate output signals. It would be possible to have the system just compute one logical instant after the other, to start with the next reaction as soon as the previous one has finished; however, typically the ticks are executed at some fixed frequency, resulting in an interval T between each tick, where T is determined by the real-time requirements of the system. The conceptually infinitely fast computation of the reaction within a logical tick, plus in the case of Esterel the unique signal presence/absence status throughout a logical tick, together constitute the *synchrony hypothesis*. In practice, the results of a logical instant of course cannot be computed infinitely fast; however, to maintain the abstraction of the synchrony hypothesis, a logical instant must be computed within some desired reaction time T.

As a short example, consider the Esterel module ABRT presented in Figure 2(a). In the *initial tick*, the first logical instant, the output signal S is emitted, and the tick is finished when control reaches the halt statement. The semantics of halt is that control never goes past it, which would mean that the program stays there throughout all subsequent ticks. However, there is also the enclosing abort statement, which states that any enclosed statements—which we call the *abort body*—are aborted when the input signal A—which in this case is the *abort signal*—is present. An exception is the first tick when the abort body is entered, in which case the abort signal is not considered yet; otherwise this would be an *immediate abort*. In the ABRT module, it is a *strong abort*, which means that in case of an abort, control does not reach the abort body. However, in this case the behavior is not different from that of a *weak abort* (which would be denoted by weak abort), where in case of an abort, the abort body can complete the execution of its logical in-

stant before control is transferred past the abort body. To summarize, ABRT emits an S in the initial tick, and then terminates in the first following tick where A is present; T is never emitted.

3.1 The KEP Instruction Set

The KEP2 employs a 32-bit wide instruction word with a separate 16-bit wide inner data bus. The KEP assembler language contains thirty instructions. The most common Esterel statements, including a majority of the reactive kernel statements, can be represented directly. Other Esterel statements can be implemented by standard Esterel syntax translation.

As discussed further in Section 3.3, the KEP implements the behavior of (weak or strong) aborts with Watcher components, which just need to be configured once at the entrance of an abort body[1]. Once an abort body is entered, it can be executed without the need for extra instructions that monitor whether an abort is triggered or not. The KEP assembler code for the ABRT example is shown in Figure 2(b). The "abort ... when A" statement of the Esterel version has been turned into an "ABORT 1, A, A0" KEP assembler instruction, which states that the following statements until the label A0 constitute an abort body that should be strongly aborted when the signal A has occurred once. At the end of the program, an additional HALT statement is inserted to provide deterministic behavior when the program terminates. The other KEP assembler statements correspond directly to the Esterel source. An exception is the first statement after the interface declaration; this statement emits the signal _TICKLEN with a value of six. This purpose of this signal is to initialize the Tick Manager with a maximal tick length, as discussed in Section 3.4.

As the KEP instruction set has been designed specifically for Esterel, the task of building a compiler that translates Esterel to KEP assembler is fairly straightforward. We have implemented such a compiler based on the Columbia Esterel Compiler (CEC) [7]. The CEC reads the Esterel input file and builds an abstract syntax tree (AST). All further transformations are based on that tree structure.

Our compiler includes a *WCRT analyzer*, which computes the maximal instruction count per logical instant and automatically generates the corresponding "emit _TICKLEN" statement. The WCRT analyzer is based on the techniques presented in Section 4, and also employs the Esterel AST. Hence it works at the level of Esterel itself, while taking into consideration how KEP assembler is generated. The WCRT analyzer also annotates the code with comments ("*T0*", "*N1.2*", etc.) that provide detailed timing information for each statement, as explained further in Section 4.1.1.

3.2 The KEP Architecture

The top-level I/O signals of the KEP2 are illustrated in Figure 3. The environment can reset the processor via the Reset pin. An external clock must be connected to the OscClk pin. ROMData and ROMAddr are data and address buses for the instruction memory. There are n_{in} pins Sin to signal

[1]KEP even allows one to move the abort initializations up to the beginning of the program, which is an optimization when abort bodies are re-entered, and our response time analysis can handle this case as well. However, to keep the presentation simple, we here do not consider this optimization further.

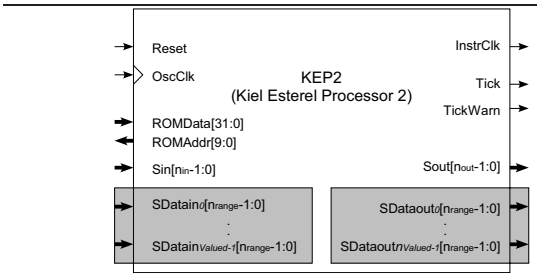


Figure 3: The KEP interface.

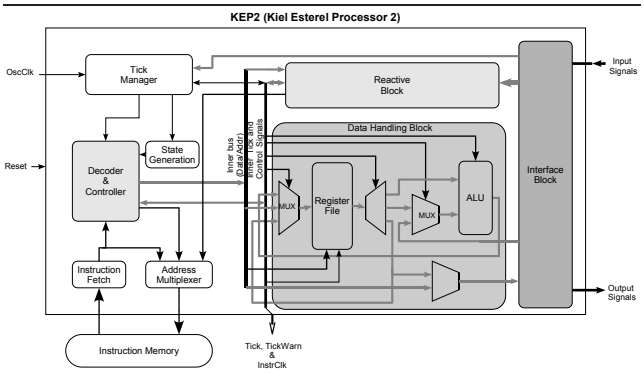


Figure 4: The KEP architecture overview.

the presence of input signals, and Valued data buses SDatain of width n_{range} to provide values for input signals. There are similar pins and buses for output signals. The InstrClk indicates the instruction clock; each instruction cycle lasts three OscClk cycles. The Tick pin indicates the logical tick of Esterel, see also Section 4.3, and the TickWarn pin is set high when a timing violation is detected, see also Section 3.4.

The architecture of the KEP2, shown in Figure 4, is inspired by the three layers that constitute a reactive program [5], *i. e.*, the *interface* layer, the *reactive kernel*, and the *data handling* layer. The Interface Block handles input reception and output production. The Reactive Core, consisting of the Decoder & Controller and the Reactive Block, decides what computations and what outputs must be generated when it reacts to inputs. The Data Handling Block performs the classical computations.

3.3 The Reactive Core

In the Reactive Core, the Reactive Block includes the AWAIT Element, implementing a blocking wait for a single signal, the CAWAIT Element, implementing concurrent waiting, and the PRESENT Element, implementing signal testing. These elements are described in detail in a separate technical report [11]; in the following, we will describe the Preemption Element, which contains a configurable number of Watcher modules that are responsible for implementing the preemption operations. Figure 5 shows a configuration with three Watcher modules. According to the Esterel semantics, a preemption (abortion or suspension) is *enabled* when control is in its body, and *disabled* when control is outside of its body. When a preemption is *enabled*, the corresponding trigger signal is watched and the module can react to the presence of

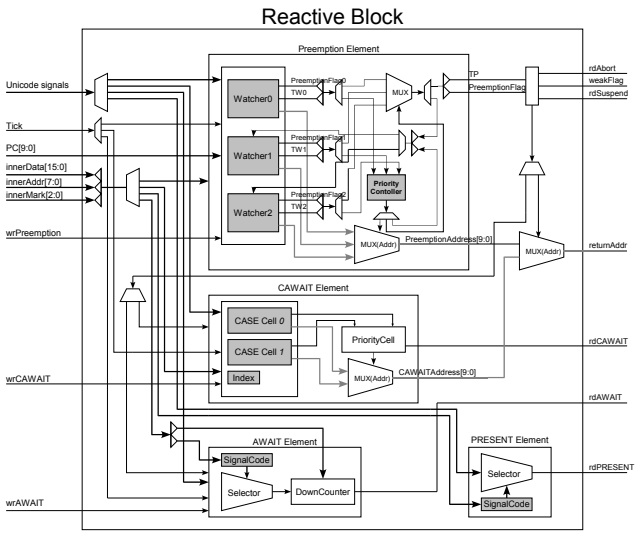


Figure 5: Architecture of a Reactive Block with three Watchers.

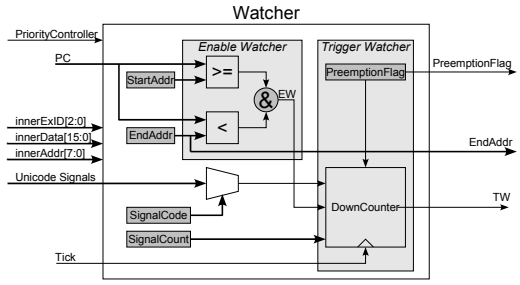


Figure 6: The structure of the Watcher.

it (is *active*). Otherwise, the signal does not cause preemption. We call this scheme *Inside/Outside Preemption Range Watching* (*IOPRW*). A Watcher, shown in Figure 6, contains two functions to implement the IOPRW, the Enable Watcher and the Trigger Watcher.

The Enable Watcher watches the program counter PC and compares it with the corresponding preemption's start and end addresses. Based on that, it decides whether this preemption should be in the *enabled state* or in the *disabled state*. If the watched signal is present on the Tick rising edge and the Watcher is in the enabled state, the Watcher triggers a corresponding action, unless it is overridden by another Watcher with higher priority, *e. g.*, an enclosing nesting activates a suspension and freezes the state of its body.

The function of the Trigger Watcher depends on the configuration of the Watcher. For an abortion, it watches the trigger signal; if the signal occurs, the watcher goes into the *triggered state* and counts down the signal count; then, depending on whether the trigger signal count specified by the abortion statement has already been reached, the watcher decides whether it should go into the *terminated state*, which would kill the abort body, or not. For a suspension, the watcher watches the trigger signal and decides whether the suspension body ought to go into the *suspended state* or not. Once an abortion is terminated or a suspension is activated, a TW event will be emitted.

3.4 The KEP Tick Manager

One of the distinguishing features of the Kiel Esterel Processor is the Tick Manager, which autonomously ensures that logical ticks are computed at a fixed frequency. Furthermore, the Tick Manager internally monitors timing violations. The Tick Manager is activated by setting the pre-defined valued signal _TICKLEN to a certain value, typically at the beginning of the program. In the ABRT example shown in Figure 2(b), the statement "EMIT _TICKLEN, #6" defines the length of a logical tick to be six instruction cycles. Hence, if a tick is finished in less than _TICKLEN instruction cycles, KEP idles for the remaining cycles before starting the next tick. If, on the other hand, a tick is not finished within _TICKLEN cycles, this is considered a *tick length timing violation*. Such timing violations are signaled to the environment via a special signal, TickWarn, with a dedicated output pin; this signal remains present until the next reset of the processor. As the KEP instruction cycles require a fixed number of clock cycles, providing a value for _TICKLEN alleviates the need for the environment to provide a timer that starts the ticks in regular intervals. Furthermore, the self-monitoring makes it easy for the environment to detect any timing violations.

Figure 7 illustrates the KEP timing behavior for another example, this time for an incorrect setting of _TICKLEN. For the module OVERRUN in Figure 7(a) and some given input scenario (input signal D always absent), the KEP produces the timing shown in Figure 7(b). In this example, the program is running on a KEP2 implemented on a Memec V2MB1000 Development Board at a rate of $T_{osc} = 41.67ns$ (24 MHz), the waveform was recorded by an Agilent 1683A Logic Analyzer. In OVERRUN, the first EMIT statement sets _TICKLEN to three; in other words, the module claims that $V_{ticklen}$, the maximal number of instructions executed within a tick, is at most three. If _TICKLEN is larger than $V_{ticklen}$, it means that the ticks last longer than is necessary to finish tick computations before the next tick starts; if _TICKLEN is smaller than $V_{ticklen}$, this means that we run the risk of timing violations. In the example, the first logical tick lasts three instruction cycles. In the second tick, the controller has to execute five instructions until the AWAIT statement is executed. Hence, the TickWarn signal will be set high when the fourth instruction cycle arrives to indicate the tick length timing violation. The goal of the WCRT analysis presented in the next section is to automatically deduce a value for _TICKLEN that is just large enough to never induce a timing violation; ideally, we achieve _TICKLEN $= V_{ticklen}$.

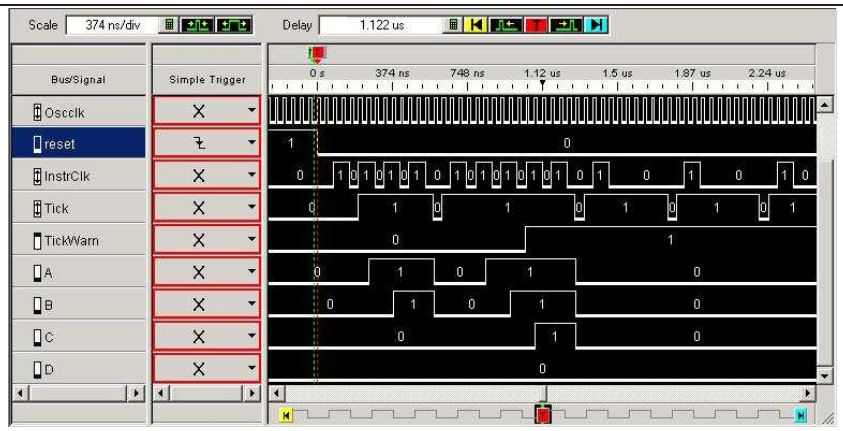
4. WORST CASE REACTION TIME ANALYSIS

The execution of translated and assembled Esterel programs on the KEP processor is done sequentially for each currently active statement. Most statements need one processor clock cycle to execute, just abort statements need an additional clock cycle. For the KEP, just as in Esterel, we distinguish statements as to whether they consume logical time or not. Most statements are considered *instantaneous*, that is, they complete within one tick and do not consume logical time; these statements include for exam-

```
% KEP Assembler
% module OVERRUN
INPUT D
OUTPUT A,B,C

EMIT _TICKLEN, #3
EMIT A
EMIT B
PAUSE
EMIT A
EMIT B
EMIT C
AWAIT D
```

(a)



(b)

Figure 7: Example KEP2 assembler code illustrating the Tick Manager (a), and resulting timing diagram (b).

ple EMIT, PRESENT, or GOTO. However, there are also *non-instantaneous* statements, which do not complete within a tick, but are resumed in the next tick; these statements include PAUSE, AWAIT, and HALT. As the non-instantaneous statements constitute the boundaries of the ticks, we also call these statements *tick delimiting* statements.

To compute the WCRT, we need to find the longest possible instruction sequence without any tick-delimiting statements in a given KEP program. As mentioned in the introduction, this is related to the static computation of the worst case execution time (WCET) for general purpose processors and compilers. However, the analysis of the maximum tick length on the KEP processor is much simpler by comparison. The architecture has been designed with execution speed in mind, but there are no features such as pipelining, caches, or out-of-order execution that break the fixed relationship between the processor clock and the instruction clock. Of at least equal importance is, however, that Esterel itself imposes tight constraints on possible control flows, thus simplifying the analysis *if* this control flow is preserved on the execution platform, as is the case with the reactive processing approach.

4.1 Constructing the KEP Assembler Graph

The AST is converted iteratively into a graph structure representing the control flow of the Esterel program. We call this graph the *KEP Assembler Graph* (*KAG*). We define the transformation of the Esterel program into the KAG as a structural induction on the AST of the Esterel program. A selection of transformation rules is given in Figure 8. We distinguish two types of KAG nodes: *Non-terminal nodes* (squares) denote an instruction (or sequence thereof) executed within a tick. In the graphical KAG representation, non-terminal nodes are inscribed with the *cost* of that node, which is the number of instruction cycles required to execute the instruction (sequence). *Terminal nodes* (small double circles) indicate the beginning/end of a tick. Terminal nodes are considered to have a cost of zero. Furthermore, we define a class of non-KAG nodes, the *working nodes* (boxes with rounded edges), that are generated as an intermediate result while transforming the AST into the KAG.

A certain difficulty when creating the KAG lies in the exceptional control flow edges introduced by abort and trap/exit statements. Those edges cannot be immediately added to the graph while analyzing a particular abort or trap statement, because those edges connect to nodes (possibly deeply) within the body of the statement, which at that moment is not yet transformed. To handle this, the edges stemming from an exceptional control flow are represented by a tuple (A, W, e) associated with each working node. A and W are the sets of nodes that are sinks of edges out of the body of enclosing abort and weak abort statements, respectively, and e is a function mapping "exit T" nodes to the corresponding enclosing "trap T" body.

The (*initial*) rule, top-left in Figure 8, defines the start of the transformation of an Esterel program p. Two terminal nodes are created, indicating the beginning and the end of the program. The program p is placed in a working node, where A, W, and e are initialized as empty sets. The working node is followed by a non-terminal node of cost 1, which represents the halt statement that is added to the end of the program in the compilation from Esterel to KEP assembler. Rule (*emit*) contains the transformation rule for a simple emit statement. That statement is translated into a node with cost 1 and unchanged incoming and outgoing nodes. In general, a working node has one successor, denoted by o, and an arbitrary number of predecessors, denoted by the set I. Hence, the edge from I to the working node in the emit rule represents an arbitrary number of edges; to indicate this, this edge is drawn as a double edge. Code blocks joined by a sequence operator (;) are transformed into two distinct blocks by the (*sequence*) rule. Those two blocks are not yet final and must be transformed further. The (*halt*) rule is a simplified version of (*await*) (see below), except that the block continuing in the next instant is missing as execution stops at the halt instruction.

Rule (*abort*) reflects the initialization costs of two clock cycles for the abort watchers, and adds an entry to edge list A for the evaluation of the abort bodies. Rule (*trap*) just adds a relation from the trap signal to the termination node to the list of e edges. Those entries are connected to nodes added for exit statements by the (*exit*) rule.

Rules (*await*) and (*pause*) generate three sequentially connected nodes. The await and pause statements end the execution of the current instant, hence a terminal node to indicate a tick boundary. In case of a weak abort, the abort body is

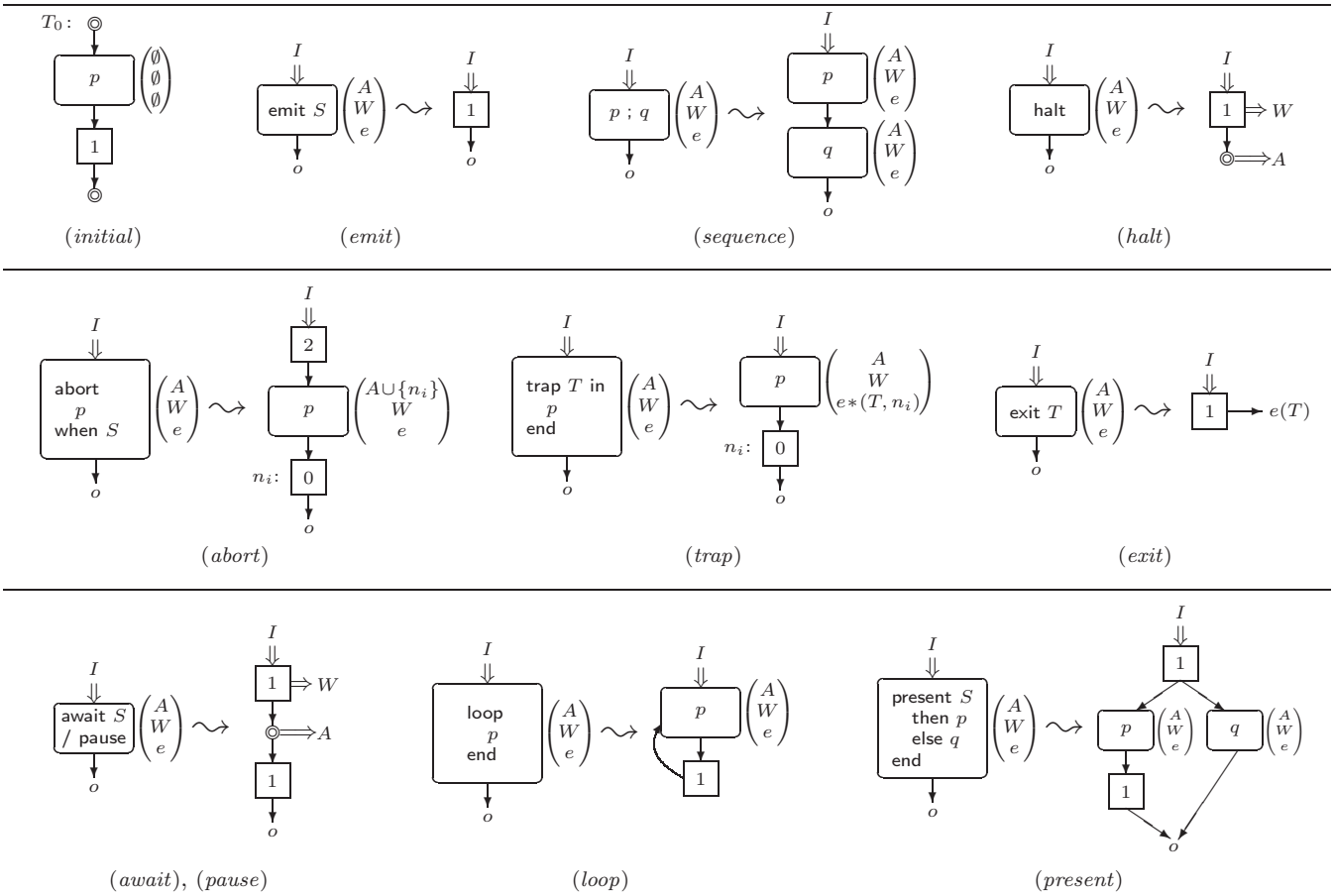


Figure 8: Transformation Rules to generate the KEP Assembler Graph.

still executed for the current instant, that is, until a terminal node is reached, before the body is aborted; therefore, the W edges are connected to the node just before the terminal node. On the other hand, a strong abort specifies that the enclosed body must be aborted immediately, from the current terminal node; hence the A edges are connected to the terminal node created in this rule. The rule also generates two non-terminal nodes of cost 1, indicating that await and pause consume one instruction cycle at the beginning as well as at the end of an instant. The transformation of loop into KEP assembler produces a GOTO statement at the end of the enclosed statement body. Hence, in rule (*loop*), a node with cost 1 connects back to the start of the loop body. The (*present*) rule is a little bit more involved. The evaluation of the branch condition is represented as an initial block with cost 1. Both the then and else branches are transformed in separate branches of the control flow graph. A trailing node with cost 1 is added to the then branch, corresponding to the GOTO statement skipping the else branch. The cost of skipping the then branch to reach the else branch is already included in the evaluation of the branch condition. There are special cases for missing then or else branches, which are omitted here—as well as several other transformation rules, for example for handling await case statements, which are a little more involved but follow the same principles as the rules presented here.

Figure 9 shows the complete transformation of the Esterel module ABRT from Figure 2 into a KAG. The initial graph produced by the (*initial*) rule is shown in Figure 9(a). For each step, the applied transformation rule is specified; for example, the (*abort*) rule transforms the initial graph into the graph shown in Figure 9(b). The iterative application of the transformation rules finally results in the KAG shown in Figure 9(g), which could serve as a basis for computing maximal instant lengths.

4.1.1 Optimizing and Labeling the KAG

Again considering the KAG shown in Figure 9(g), we note that there are some straightforward reduction rules that result in a for our purposes equivalent, but smaller KAG. First, combining adjoining nodes results in the KAG shown in Figure 9(h). Then, eliminating dead code results in the KAG shown in Figure 9(i). In this final KAG, which is the graph computed by our implementation, nodes are also annotated with a $\langle type \rangle\langle index \rangle$ label, where $\langle type \rangle$ indicates whether a node is a terminal node ("T") or a non-terminal node ("N"), and $\langle index \rangle$ is a unique index for each node. The KAG generator also uses these labels to annotate the KEP assembler code with detailed timing information for each statement, as shown in Figure 2(b). For each KAG terminal node, the corresponding non-instantaneous KEP assembler instructions that lead to this node are indicated in a comment; for ex-

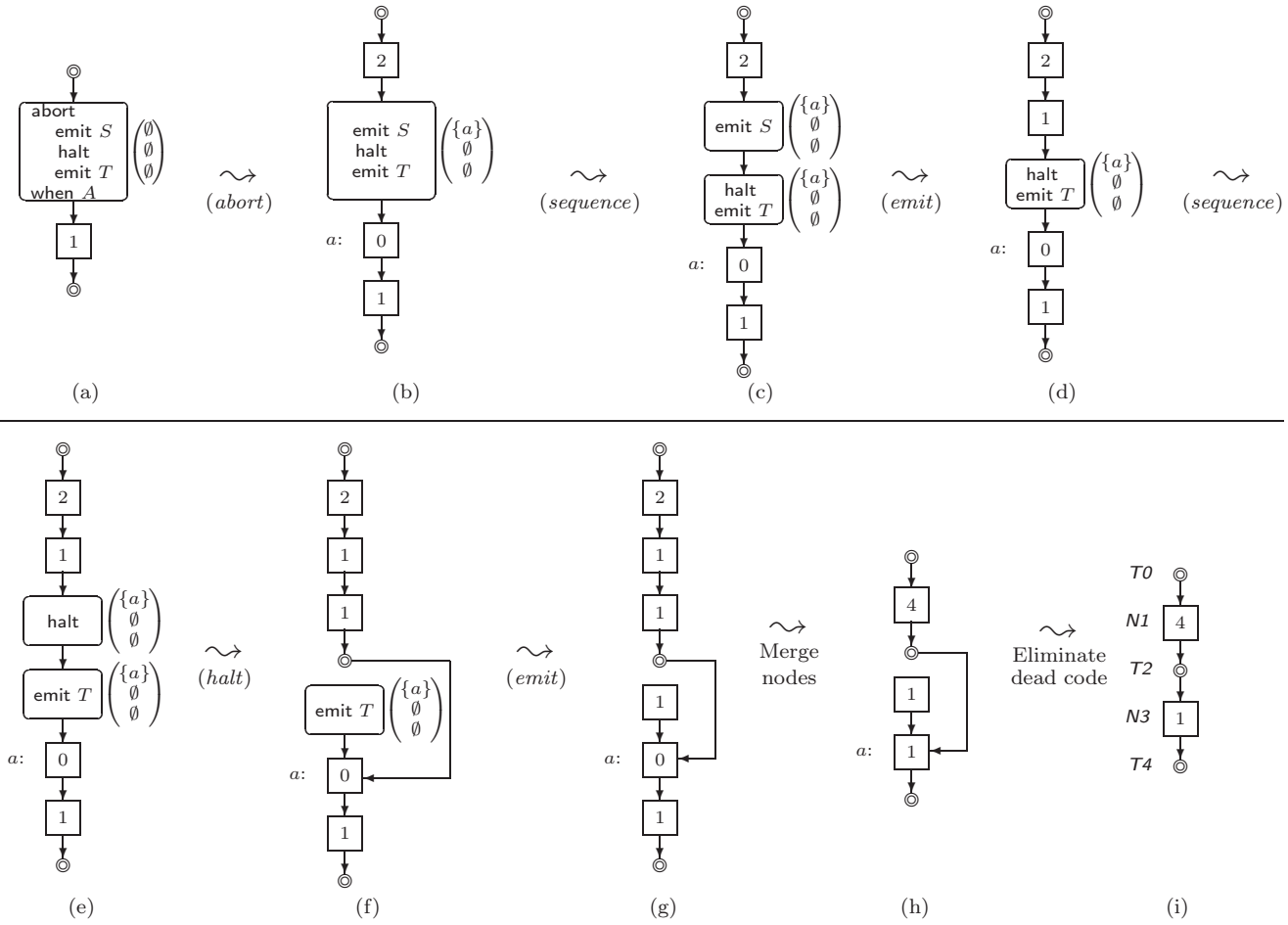


Figure 9: Transformation of Esterel module ABRT (see Figure 2) into a KEP Assembler Graph.

ample, the terminal node *T2* corresponds to the first HALT instruction. For each non-terminal node, the corresponding KEP assembler statements accumulated into this node are annotated with a $\langle type \rangle \langle index \rangle . \langle count \rangle$ comment, where $\langle count \rangle$ indicates the accumulated instruction count for that node; for example, the first HALT statement brings the accumulated instruction count of *N1* to four.

4.1.2 The ATM Example

As a slightly more involved example, consider the Esterel program in Figure 10(a) (see Roop *et al.* [18]), which implements the basic control structure for an automatic teller machine (ATM). The program waits for the card being inserted, the PIN number entered, and the action selected. The procedure is aborted and restarted when an error condition occurs, such as an unreadable card or the wrong PIN.

Figure 10(b) contains the ATM program translated into KEP assembler code. The WCRT analyzer has inserted a statement after the INPUT/OUTPUT definitions that emits _TICKLEN with a value of eight, meaning that the maximum instant length has been determined to be eight clock cycles long. The timing analyzer has also annotated the code with low-level timing information, relating to the generated KAG, which is shown in Figure 10(c). Straight edges in the graph from node to node denote the sequential control flow, arcs result from weak abort, abort, await or goto statements. The path $\langle$*T4*, *N5*, *N18*, *N1*$\rangle$ is decorated with dashed lines because it is a longest path. Longest paths are not necessarily unique; here, the path $\langle$*T4*, *N5*, *N17*, *N18*, *N1*$\rangle$ is also a path of maximal cost.

4.2 Finding Longest Paths in the KAG

We define the *length* of a path within a KAG to be the sum of the costs of all nodes on that path. To compute the maximal number of instruction cycles consumed within an instant, we must compute the maximal length of all paths between any pair of instant-end nodes (which we also called terminal nodes, see Section 4.1) in the KAG. Given a KAG, let N be the set of all nodes, let $T \subseteq N$ be the set of terminal nodes, and let E be the set of edges. One could apply Dijkstra's algorithm for longest paths on every terminal node, and would obtain the maximum cost for every pair of nodes $(t, n) \in T \times N$. However, this is clearly overhead, as it would consider all paths irrespective of whether they cross tick boundaries or not. Instead, a KAG may be partitioned into subgraphs $G_1, \ldots, G_{|T|}$, where each G_i has a unique source node from the set of terminal nodes T. Furthermore, valid Esterel programs must not contain cycles within an instant; in fact, our algorithm can also be used to detect such cycles, as in that case it sets the maximum path length to

```
module ATM:

input
  cardInserted, pinEntered,
  sumEntered, transactionOK,
  incorrectPin, invalidCard,
  withdraw, checkBalance;
output
   insertCard, enterPin,
   selectOption, processTransaction,
   releaseSum, printReceipt,
   ejectCard;

loop
  emit insertCard;
  await cardInserted;
  weak abort
    weak abort
      emit enterPin;
      await pinEntered;
      emit selectOption;
      await
        case withdraw do
          await sumEntered;
          emit processTransaction;
          await transactionOK;
          emit releaseSum;
          emit printReceipt;
        case checkBalance do
          emit processTransaction;
          await transactionOK;
          emit printReceipt;
      end await
    when incorrectPin
  when invalidCard;
  emit ejectCard
end loop

end module
```

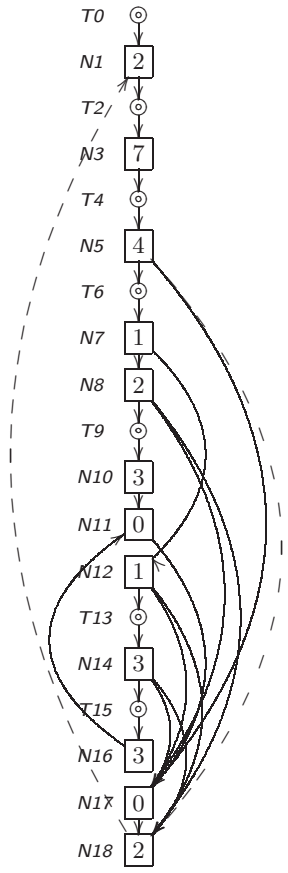
(a)

```
INPUT
  cardInserted, pinEntered, sumEntered,
  transactionOK, incorrectPin, invalidCard,
  withdraw, checkBalance;
OUTPUT
  insertCard, enterPin, selectOption,
  processTransaction, releaseSum,
  printReceipt, ejectCard;

EMIT _TICKLEN, #8

A0:                              % T0
   EMIT insertCard               % N1.1
   AWAIT cardInserted            % N1.2, T2, N3.1
   WABORT 1, invalidCard, A1     % N3.3
   WABORT 1, incorrectPin, A2    % N3.5
     EMIT enterPin               % N3.6
     AWAIT pinEntered            % N3.7, T4, N5.1
     EMIT selectOption           % N5.2
     CAWAIT withdraw, A3         % N5.3
     CAWAITE checkBalance, A4    % N5.4, T6, N7.1
A4:                              % N8.0
        EMIT processTransaction  % N8.1
        AWAIT transactionOK      % N8.2, T9, N10.1
        EMIT printReceipt        % N10.2
        GOTO A31                 % N10.3
A3:                              % N12.0
        AWAIT sumEntered         % N12.1, T13, N14.1
        EMIT processTransaction  % N14.2
        AWAIT transactionOK      % N14.3, T15, N16.1
        EMIT releaseSum          % N16.2
        EMIT printReceipt        % N16.3
A31:                             % N11.0
A2:                              % N17.0
A1:                              % N18.0
   EMIT ejectCard                % N18.1
   GOTO A0                       % N18.2
```

(b)



(c)

Figure 10: Automatic Teller Machine: Esterel program (a), KEP assembler (b), KEP Assembler Graph (c).

∞. To summarize, finding the longest path in the KAG can be reduced to the problem of detecting cycles in a graph and finding the longest path in a set of directed acyclic graphs.

The resulting algorithm is given in Figure 11. Each node n has several attributes. The *terminal* flag indicates whether n is a terminal node. The *cost* is the number of clock cycles the node n needs to execute; for terminal nodes it is $cost = 0$. The set *succs* contains the departing edges connecting n to other KAG nodes. The *visited* flag is set if n has already been visited in the algorithm. The integer *len* is the maximum instant length counted from n on; that is, the longest path originating from n that does not contain any terminal nodes (except n, if n is a terminal node).

Given a set of nodes N of a KAG, the function *max_ instant_ len* computes the longest path in the KAG. After some initializations, the maximum of all paths originating from n that do not contain any other terminal nodes is computed for each terminal node n, using the helper function *get_ len*. Then, the maximum of these maxima is built and returned by *max_ instant_ len*.

The function *get_ len* explores the subgraph originating at n. It first checks whether *len* has already been computed. If so, *len* is returned and we are done. Otherwise, it checks whether n has already been visited. If this is the case, we have encountered an instantaneous loop, which is forbidden in Esterel; we set *len* to infinity and are done. Otherwise, we perform a depth-first traversal of the subgraph originating in n, recursively calling *get_ len* for all successors of n, and setting $n.len$ to the sum of the *cost* of n plus the maximum of the lengths of the paths originating in the successors of n. As it turns out, the complexity of the algorithm is $\mathcal{O}(|N|+|E|)$, which is the best one can hope for. The proof of this, which is overall fairly straightforward, involves showing that there are $\mathcal{O}(|E|)$ calls to *get_ len* and that there is an amortized constant cost to each such call.

4.3 Bounding Concrete Reaction Times

To understand how a given value for _TICKLEN translates to concrete reaction times, we now consider the KEP instruction timing and signal sampling. Let T_{osc} be the basic clock rate supplied to the processor (via the Oscclk pin). The KEP signals the passage of logical ticks to the environment via a Tick signal, with a corresponding output pin. As illustrated in Figure 12, the KEP samples its inputs at rising edges of Tick. Furthermore, the KEP holds all outputs generated during a tick until the end of the tick, so the falling edges of Tick indicate when the environment should

```
1  int max_instant_len(Nodes N) {
2
3    forall n ∈ N do {
4      n.visited := false
5      n.len := ⊥
6    }
7    T := {n ∈ N | n.terminal}
8    return max_{n∈T} get_len(n)
9  }
10
11 int get_len(Node n) {
12   if (n.len = ⊥) {
13     if (n.visited) {
14       n.len := ∞          // Instantaneous loop
15     }
16     else {
17       n.visited := true
18       n.len := n.cost + max_{s∈(n.succs\T)} get_len(s)
19   }}}
20   return n.len
21 }
```

Figure 11: Finding the maximum instant length.

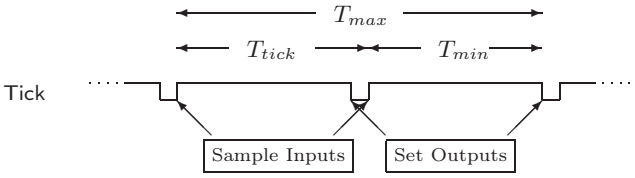


Figure 12: A waveform of the Tick signal and derived values.

sample the outputs generated by the KEP[2]. After the falling edge, Tick remains low for a gap of width T_{osc} before it rises again. Let T_{react} be the time from the occurrence of an input signal (or combination thereof) until the generation of a corresponding output signal (or combination thereof). Assuming that the environment may generate inputs at arbitrary times and that the KEP is computing reactions on its own, regular schedule, T_{react} may vary within an interval:

$$T_{min} \leq T_{react} < T_{max}. \quad (1)$$

Here, the lower bound T_{min} is determined by the time it takes to actually compute a reaction, *i. e.*, the length of a tick—which is the time from a rising edge of Tick to a falling edge of Tick. This reflects the case where an input is sampled *immediately* when it is generated; *i. e.*, an input happens to occur just when the KEP starts to compute a reaction. Let $V_{ticklen}$ be the maximal number of instructions that must be executed to compute a tick. As each KEP instruction takes three clock cycles and an additional cycle is introduced at each tick, the lower bound on the reaction time can be computed as follows:

$$T_{min} = (3V_{ticklen} + 1) * T_{osc}. \quad (2)$$

The upper bound T_{max} reflects the case when an input occurs just after inputs have been sampled, in which case the sampling of this input is delayed by the length to compute a tick, given by T_{min}, plus the gap T_{osc}. We denote the interval from one rising edge of Tick to the next rising edge by T_{tick} and obtain:

$$T_{tick} = T_{min} + T_{osc}, \quad (3)$$
$$T_{max} = T_{min} + T_{tick} \quad (= (6V_{ticklen} + 3) * T_{osc}). \quad (4)$$

For example, we obtain for a clock rate of $T_{osc} = 50ns$ and $V_{ticklen} = 8$ (as in the ATM example considered earlier) the following range for the reaction times:

$$1.3\mu s \leq T_{react} < 2.55\mu s. \quad (5)$$

[2]Actually, it is a little more complicated; to avoid a race condition between the sampling of the outputs and the outputs becoming low again, the output signals are latched at each falling edge of InstrClk, and then propagated to the environment at the falling edge of Tick.

Table 1: Comparison of mca200 CURVE module implementations; code sizes and RAM usages are given in words/bytes.

	KEP2-B (16-bit)	Hardware (16/32-bit)	MCS51 (8-bit)	Microblaze (32-bit)
Logic Cells	1326	968/1510	-	1906
Code size	185/740	-	1070/1636	436/1744
RAM Usage	9/18	-	31/31	19/76

Table 2: Comparison of the codes sizes, in words. On the Microblaze, one word equals four bytes.

Module name	Esterel lines	Microblaze (words) V5	V7	CEC	KEP2 (words)
SPEED	11	276	1081	253	11
BELT_CONTROL	14	440	1169	340	18
TIMER	6	368	1160	295	9
CONTROLLER	26	560	1226	487	24
DEBOUNCE	31	392	1198	299	28
ALARM_COMPARE	16	315	1109	265	14
SPEEDOMETER	23	328	1145	293	20
DASHBOARD_TIMER	77	617	1388	541	65
FRC	26	375	1163	313	18
CURVE	190	1307	2017	436	185
BAT_DIAG	45	487	1274	378	63
VER_ACC_DIAG	38	433	1229	303	41
LONG_SPEED_STRAT	60	573	1306	319	56

5. EXPERIMENTAL RESULTS

To quantitatively compare the data handling abilities between the Esterel processor and other implementations, we used the CURVE module from the mca200 test bench [8] as an example, since it is a typical module that includes varied data handling statements. Table 1 compares the resource usages of the KEP2 with different hard- and software implementations. For the hardware implementations, we synthesize the module to VHDL with the Esterel V7 compiler, since other hardware compilers cannot support valued signals. The V7 compiler does not provide a data ranging function, *i. e.*, an integer type valued input signal will always occupy a 32-bit bus to represent the carried value. As an optimization, we manually resized all of the valued signals and variables to 16-bit width, since that range is sufficient for this Esterel module. Then those VHDL programs are implemented by the ISE6.3.03i, and the speed (default) optimization is used. For the software implementation, we use the CEC V0.3 compiler to synthesize the module to a C program, which is then compiled onto the MCS51 (using the Keil C51 compiler V6.12 with the default—level 8—

Table 3: Comparison of sizes and clock rates between the KEP2 series and RePIC.

	RePIC	KEP2-A	-B	**-C**	-D	-E	-F	-G	-H	-I	-J
Number of CAWAIT elements	2	2	2	**2**	4	2	2	2	2	2	4
Number of Watchers	4	2	2	**4**	4	6	4	4	4	4	6
Counter Value Range	1	1	1	**1**	1	1	255	1	1	1	255
Inputs/Outputs	12/12	8/8	8/8	**12/12**	12/12	12/12	12/12	24/24	12/12	12/12	24/24
Valued Inputs/Outputs	1/1	2/2	2/2	**1/1**	1/1	1/1	1/1	1/1	4/4	1/1	4/4
Datapath Width	8	8	16	**8**	8	8	8	8	8	16	16
Logic Cells	2068	998	1326	**1476**	1564	1898	1582	1756	1626	1714	2810
Max Osc Freq (MHz)	40.27	54.80	45.06	**42.87**	42.87	39.46	43.19	42.59	44.43	43.00	37.85
Instruction Freq (MHz)	10.1	18.27	15.02	**14.29**	14.29	13.15	14.4	14.2	14.81	14.33	12.62

optimization), a classical widely used 8-bit processor, and the 32-bit Microblaze soft processor core (compiled by gcc for Microblaze version 2.95.3-4, using the default—level 2—optimization). Note that the lengths of MCS51's instructions vary; here, a *word* represents a complete assembler line. We notice that Logic Cell count of the KEP2 processor is significantly smaller than that of the Microblaze, and is comparable with a hardware implementation of the CURVE module. We also note that both the code size and the RAM usage are significantly smaller than those of the software implementations.

To further illustrate the compactness of the KEP2 code, Table 2 compares executable code sizes between the KEP implementation and the Microblaze software implementation for some other benchmarks. Table 1 revealed that the code for the 32-bit Microblaze is smaller (in words) than that of the 8-bit MCS51; therefore, we use the Microblaze as reference point. The programs are standard test cases from the literature [3, 1, 8], which not only contain reactive control flow, but also include arithmetic and logical data handling. The module is first translated into the KEP assembler program and then compiled to the KEP executable codes. This is then compared with software synthesis results of the Esterel Compiler V5.92, the Esterel Compiler V7 and the CEC compiler 0.3 for the Microblaze. We notice that the optimized data path of the KEP results on average in a 89% reduction of codes size when compared with the best result of the Microblaze implementation. In fact, the Esterel module's line count is very close to the KEP2's codes size (in words). This fact implies that the KEP handles the Esterel statements on a high level. Practically, the majority of Esterel statements can be translated into KEP assembler instructions word by word.

As mentioned in the introduction, the KEP has been designed to be highly configurable. Table 3 compares ten different KEP2 variants which include different elements to target various applications. The KEP2-C is the configuration closest to the RePIC; however, RePIC uses four clock cycles to execute an instruction cycle, but the KEP2 uses only three clock cycles, and the KEP2 typically takes significantly less instructions to implement the same behavior, as illustrated earlier.[3]

[3]Regarding the logic cell count, one should note that the KEP2's implementation is based on a Xilinx's XC2S100-6TQ144 FPGA chip, and the RePIC is implemented on an ALTERA' EP20K200EFC484-2 FPGA chip. However, the basic units of those two chips have similar structures, functions, and speed; for example, an implementation of a CQPIC [16] soft core (the original processor core of RePIC) on the ALTERA chip uses 1082 logic cells and runs at 45.83 MHz, whereas on the Xilinx chip it needs 1156 logic cells and runs at 48.47 MHz.

6. CONCLUSIONS AND OUTLOOK

This paper presented the Kiel Esterel Processor, a semi-custom, configurable Esterel processor. It consists of a Reactive Core and an optimized data path for the direct execution of Esterel programs. The KEP supports Esterel preemption statements in a very precise, direct and efficient way, and allows their arbitrary combination and nesting. The maximal nesting depth of these constructs is only limited by the number of watchers that are provided by the KEP; however, this number is configurable for a particular KEP. The KEP supports valued signals and signal counters, local signal declarations, and the pre operator. A technical report [11] describes this in further detail, along with a description of other architectural features and the complete instruction set.

The KEP2 already provides true concurrency in that it can watch several trigger signals simultaneously; for example, it does not slow down with increasing abort nesting depth. However, the KEP2 does not implement Esterel's concurrency operator ("||"). To handle this, the KEP2 could be extended into a multiprocessor architecture [9]; another alternative that we are currently exploring is interleaved execution. Other improvements concern the direct implementation of further Esterel constructs, such as the immediate signal triggering, which can already be handled by the current architecture, but require multiple instructions to do so.

We have also presented a method to determine the WCRT of Esterel programs in the context of the KEP reactive processor. This analysis is more precise than the timing analysis of sequential programs running on traditional processors. To a certain extent this has been made possible by the absence of advanced architectural features such as caching and pipelining. However, what we see as even more relevant here is the synchronous execution model of Esterel and the direct implementation of this execution model on a reactive processor. To compute longest reaction paths, we have introduced the KEP Assembler Graph (KAG); based on this graph, a longest-path search, with complexity linear in the size of the KAG, determines the maximal number of instructions within an instant. This is implemented in a WCRT analyzer embedded in a compiler that translates Esterel to KEP assembler; the WCRT analyzer adds an assembler statement to initialize the _TICKLEN signal, which is used by the Tick Manager of the KEP to generate logical ticks at a constant, maximal rate. The efficiency of the WCRT analysis would also make it feasible to use it to guide an optimizing compiler. The analysis presented here should be useful for devices embedded in an environment with hard real-time demands, and where one cannot afford generous reserves of processor performance. We believe that the efficiency of this analysis and the achievable accuracy underline

the value of the synchronous programming paradigm and of the reactive processing approach.

A natural extension of the WCRT analysis would be to consider concurrency implemented by interleaving or multiprocessing. Furthermore, the KAG construction algorithm is still conservative in that all control flows possible in the Esterel program are represented, but there may exist paths in the KAG that cannot occur in the program. Hence, a maximum instant length computed from the KAG could be pessimistic. For example, it is not considered that in a non-immediate abort, the body cannot be aborted in the same instance when it is entered. This reflects the general difficulty that we are performing a structural analysis of the control flow, and do not consider actually reachable paths. It is not clear yet how much of a limitation this is in practice, as we would expect that well-written Esterel programs do not contain many unfeasible paths within an instance. Nevertheless, a natural extension to the algorithm presented here would be to consider data dependencies as well, for example using model checking.

7. ACKNOWLEDGMENTS

This work has profited from discussions with Stephen Edwards and Michael Mendler. Claus Traulsen and Ervan Darnell gave advice on earlier versions of this paper. The authors would also like to acknowledge the helpful comments from Tim Sherwood from the CASES Program Committee and from the anonymous referees.

8. REFERENCES

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

[2] G. Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[3] G. Berry. *The Esterel v5 Language Primer*. Draft Book, 1999.

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

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (EUROMICRO-RTS 2000)*, 2000.

[7] CEC: The Columbia Esterel Compiler. http://www1.cs.columbia.edu/~sedwards/cec/.

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

[9] M. W. S. Dayaratne, P. S. Roop, and Z. Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Apr. 2005.

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

[11] X. Li and R. v. Hanxleden. KEP2 (Kiel Esterel Processor 2): The Esterel Processor. Technischer Bericht 0506, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Apr. 2005.

[12] X. Li and R. v. Hanxleden. The Kiel Esterel Processor - a semi-custom, configurable reactive processor. In S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/159 [date of citation: 2005-07-18].

[13] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 196–203, Munich, Germany, March 2003. IEEE Computer Society.

[14] G. Logothetis, K. Schneider, and C. Metzler. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. In *Forum on Design Languages (FDL)*, Frankfurt, Germany, 2003. Kluwer.

[15] S. Malik, M. Martonosi, and Y.-T. S. Li. Static timing analysis of embedded software. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 147–152. ACM Press, 1997.

[16] S. Morioka. CQPIC: PIC micro computer free soft IP. http://www02.so-net.ne.jp/~morioka/cqpic.htm.

[17] P. S. Roop, Z. Salcic, M. Biglari-Abhari, and A. Bigdeli. A New Reactive Processor with Architecture Support for Control Dominated Embedded Systems. In *IEEE International Conference on VLSI Design*, pages 189–194. IEEE CS Press, Jan. 2003.

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