Worst Case Reaction Time Analysis of Concurrent Reactive Programs

Marian Boldt Claus Traulsen Reinhard von Hanxleden

Dept. of Computer Science
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{*mabo,ctr,rvh*}*@informatik.uni-kiel.de*

Abstract

Reactive programs have to react continuously to their inputs. Here the time needed to react with the according output is important. While the synchrony hypothesis takes the view that the program is infinitely fast, real computations take time. Similar to the traditional *Worst Case Execution Time* (WCET), the *Worst Case Reaction Time* (WCRT) of a program determines the maximal time for one reaction.

In this paper, we present an algorithm to determine the WCRT of a program written in the synchronous language Esterel. This value gives an upper bound for the execution time when the program is executed on a reactive processor. Specifically, we consider the execution of the Esterel program on the *Kiel Esterel Processor* (KEP), a reactive processor that can execute Esterel-like instructions. Here the WCRT directly determines an upper bound on the instruction cycles per logical tick. The WCRT also gives a guideline for the execution time when the Esterel program is compiled to software by a simulation-based approach.

We have implemented the WCRT analysis algorithm as part of an Esterel compiler for the KEP and have measured an accuracy of analysis results of about 40% on average.

Key words: Synchronous Languages, Esterel, Worst Case Execution Time, Worst Case Reaction Time, Instantaneous Reachability

1 Introduction

Many embedded systems belong to the class of *reactive systems*, which continuously react to inputs from the environment by generating corresponding outputs. For these systems, exact timing information or at least an upper bound of the execution time is crucial. To perform an exact *Worst Case Execution Time* (WCET) analysis is difficult, and in general not possible for Turing-complete languages. It 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 [18,5]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict how much time exactly 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. 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. To summarize, performing conservative yet tight WCET analysis appears by no means trivial and is still an active research area.

One step to make WCET analysis of reactive applications more feasible is to choose a programming language that provides direct, predictable support for reactive control flow patterns. One suitable candidate for this is the synchronous language Esterel [2], which has been developed for programming control-oriented, embedded systems. It directly supports concurrency and multiple forms of preemption. Based on the *synchrony hypothesis*, it offers determinism even for concurrent components. The execution of Esterel programs is divided into (logical) ticks, each of which conceptually takes no time. Esterel forbids programs with a potentially unbounded number of statements to be performed within a tick. This is reflected in the rule that there cannot be *instantaneous loops*; within a loop body, each statically feasible path must contain at least one tick-delimiting instruction. The restricted nature of Esterel and its sound mathematical semantics allow formal analysis of Esterel programs and make the computation of a WCET for Esterel programs achievable.

In addition to choosing a suitable programming language, the feasibility of WCET analysis crucially depends on the execution platform. A relatively new approach for control-oriented reactive-systems are *reactive processors* [22,14,15]. These processors directly support reactive control flow, such as preemption and concurrency. In this paper we will use the *Kiel Esterel Processor* (KEP), a reactive processor based on the synchronous language Esterel, to show that timing analysis is practical for reactive processors, hence making the reactive processing approach particularly well suited for hard real-time systems. There are two main factors that contribute to this, on the one hand the synchronous execution model of Esterel, and on the other hand the direct implementation of this execution model on a reactive processor. Furthermore, reactive processors are not designed to optimize (average) performance for general purpose computations, and hence do not have a hierarchy of caches, pipelines, branch predictors, etc. This leads to a simpler design and execution behavior and further facilitates WCET analysis.

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. Therefore we call this analysis a *Worst Case Reaction Time* (WCRT) analysis. The WCRT determines the maximal rate for the interaction with the environment. Whether WCRT can be formulated as a classical WCET problem or not depends on the implementation approach. If the implementation is based on sequentialization such that there exist two dedicated points of control at the beginning and the end of each reaction, respectively, then WCRT can be formulated as WCET problem; this is the case, for example, if one "automaton function" is synthesized, which is called during each reaction. If, however, the implementation builds on a concurrent model of execution, where each thread maintains its own state of control across reactions, then WCRT requires not only determining the maximal length of pre-defined instruction sequences, as in WCET, but one also has to analyze the possible control point pairs that delimit these sequences. Thus, WCRT is more elementary than WCET in the sense that it considers single reactions, instead of whole programs, and at the same time WCRT is more general than WCET in that it is not limited to pre-defined control boundaries.

The contribution of this paper is a WCRT analysis of complete Esterel programs including concurrency and preemption. The analysis computes the WCRT in terms of KEP instruction cycles, which roughly match the number of executed Esterel statements. As part of the WCRT analysis, we also present an approach to calculate potential instantaneous paths, which may be used in compiler analyses and optimizations that go beyond WCRT analysis.

In the following section, we consider related work. In Section 3 we will give an introduction into the synchronous model of computation for Esterel and the KEP. We outline the generation of a *Concurrent KEP Assembler Graph* (CKAG), an intermediate graph representation of an Esterel program, which we use for our analysis. Section 4 explains our algorithm in detail, while Section 5 gives experimental results, comparing the computed number of reactions with values obtained from exhaustive simulation. The paper concludes in Section 6.

2 Related Work

As mentioned in the introduction, there exist numerous approaches to classical WCET analysis. For a survey see, *e. g.*, Puschner and Burns [20]. These approaches usually consider (subsets) of general purpose languages, such as C, and take informations on the processor designs and caches into account.

Regarding the analysis of synchronous programs, Logothetis, Schneider and Metzler [16,17] 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 are 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 [12]. We here instead are interested in how long it may take to compute a single tick, which can be considered an orthogonal issue.

One important problem that must be solved when performing WCRT analysis for Esterel is to determine whether a code-segment is reachable instantaneously or delayed or both. This is related to the well-studied property of *surface* and *depth* of an Esterel program, *i. e.*, to determine whether a statement is instantaneous reachable or not, which is also important for schizophrenic Esterel programs [2]. This was addressed in detail by Tardieu and de Simone [23]. They also point out that an exact analysis of instantaneous reachability has NP complexity. We, however, are not only interested whether a statement can be instantaneous, but also whether it can be non-instantaneous.

Beside being executed on a reactive processors, Esterel programs can be synthesized to hardware [1] or compiled into software, *e. g.*, C-code; see Edwards [10] for an overview. Currently, the most efficient compilation schemes are simulation based [9,7,19,11]: the Esterel program is organized according to some kind of graphical structure and its current state is stored in a data-structure on the application level, *e. g.*, a bit-vector. Based on this vector, the current actions in the graph are triggered. While this approach produces fairly efficient code, both in size and in execution speed, it removes much of the structure from the Esterel-program, making the WCET analysis as hard as for "normal" C programs.

Ringler [21] considers the WCET analysis of C code generated from Esterel. But his approach is only feasible for the generation of circuit code [2], which scales well for large applications, but tends to be slower than the simulation based approach.

Li *et al.* [14] compute a WCRT of sequential Esterel programs directly on the source code. However, they did not address concurrency, and their source-level approach could not consider compiler optimizations. We perform the analysis on an intermediate level after the compilation, as a last step before the generation of assembler code. This also allows a finer analysis and decreases the time needed for the analysis.

The KEP contains a *TickManager* [14], which monitors how many instructions are executed in the current logical tick. To minimize jitter, a maximum number of instructions for each logical tick can be specified. If the current tick needs less instructions, the start of the next tick is delayed. If the tick needs more instructions, an error-output is set. Hence a tight, but conservative upper bound of the maximal instructions for one tick is of direct value for the KEP. See Li *et al.* [14] for details on the relation between the maximum number of instruction per logical tick and the physical timing constraints from the environment perspective.

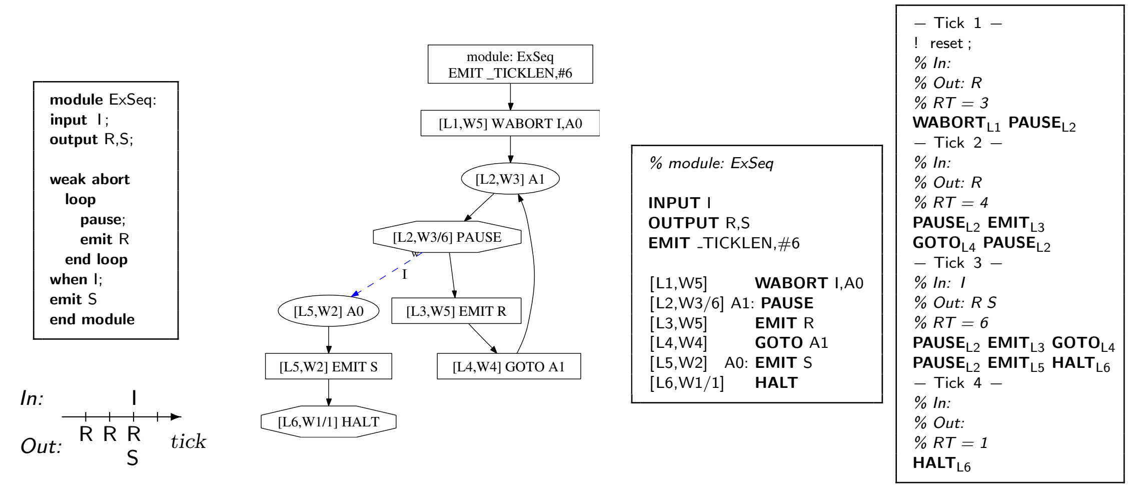
3 Esterel, KEP and the CKAG

Next we give a short overview of Esterel and the KEP. While our analysis is implemented in the compiler from Esterel to the KEP assembler, it is also of interest for other execution forms of Esterel. The analysis itself is performed on a graph representation of Esterel-programs, the CKAG.

3.1 Esterel

The execution of an Esterel program is divided into logical *instants*, or *ticks*, and communication within or across threads occurs via *signals*; at each tick, a signal is either *present* (emitted) or *absent* (not emitted). Esterel statements are either *transient*, in which case they do not consume logical time, or *delayed*, in which case execution is finished for the current tick. Per default statements are transient, and these include for example emit, loop, present, or the preemption operators. Delayed statements include pause, (non-immediate) await, and every. Esterel's parallel operator, ||, groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated.

Esterel offers two types of preemption constructs. An *abortion* kills its body when an abortion trigger occurs. We distinguish *strong* abortion, which kills its body immediately (at the beginning of a tick), and *weak* abortion, which lets its body receive control for a last time (abortion at the end of the tick). A *suspension*



```
module ExSeq:
input I;
output R,S;

weak abort
  loop
    pause;
    emit R
  end loop
when I;
emit S
end module
```

```
% module: ExSeq

INPUT I
OUTPUT R,S
EMIT _TICKLEN,#6

[L1,W5]        WABORT I,A0
[L2,W3/6] A1:  PAUSE
[L3,W5]        EMIT R
[L4,W4]        GOTO A1
[L5,W2]   A0:  EMIT S
[L6,W1/1]      HALT
```

```
— Tick 1 —
! reset ;
% In:
% Out: R
% RT = 3
WABORT$_{L1}$ PAUSE$_{L2}$
— Tick 2 —
% In:
% Out: R
% RT = 4
PAUSE$_{L2}$ EMIT$_{L3}$
GOTO$_{L4}$ PAUSE$_{L2}$
— Tick 3 —
% In: I
% Out: R S
% RT = 6
PAUSE$_{L2}$ EMIT$_{L3}$ GOTO$_{L4}$
PAUSE$_{L2}$ EMIT$_{L5}$ HALT$_{L6}$
— Tick 4 —
% In:
% Out:
% RT = 1
HALT$_{L6}$
```

(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) KEP sample trace

Fig. 1. A sequential Esterel example. The body of the KEP assembler program (without interface declaration and initialization of the TickManager) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles.

freezes the state of a body in the instant when the trigger event occurs.

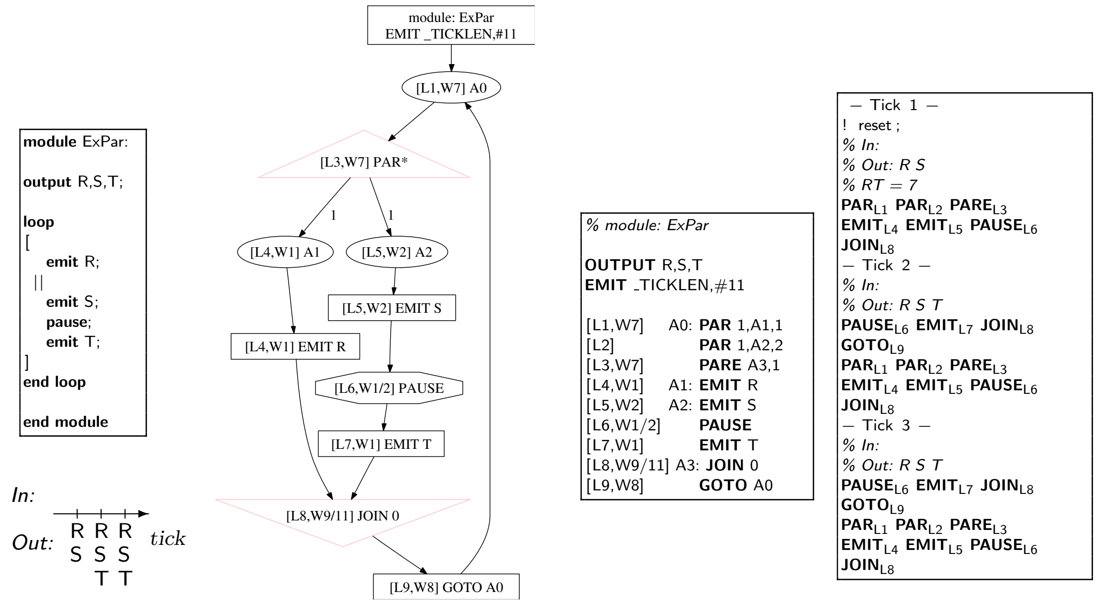
Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* (*raised*) with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are further rules when traps are nested or when the trap scope includes concurrent threads. If one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent threads are weakly aborted; if concurrent threads execute multiple exit instructions in the same tick, the outermost trap takes priority.

A simple sequential Esterel example ExSeq can be found in Figure 1(a). From the second instant on it will continuously emit the signal R. When the input I occurs, it emits R one last time. In the same instant, it also emits S and terminates. This behavior can also be observed in the trace in Figure 1(a), where input I occurs in the third tick.

For another example, consider ExPar shown in Figure 2(a), which loops over two parallel threads. The program emits the signals R and S in the first instant, and since the loop instantaneously restarts its body, it will from the second instant on continuously emit all three signals R, S, and T.

3.2 The Kiel Esterel Processor

The instruction set of the KEP is very similar to the Esterel language. The Esterel language distinguishes *kernel statements* (*e. g.*, emit, pause) and *derived statements* (*e. g.*, await, every) [3]. Derived statements are in general just syntactic sugar and



```
module ExPar:

output R,S,T;

loop
[
    emit R;
  ||
    emit S;
    pause;
    emit T;
]
end loop

end module
```

```
% module: ExPar

OUTPUT R,S,T
EMIT _TICKLEN,#11

[L1,W7]     A0: PAR 1,A1,1
[L2]            PAR 1,A2,2
[L3,W7]         PARE A3,1
[L4,W1]     A1: EMIT R
[L5,W2]     A2: EMIT S
[L6,W1/2]       PAUSE
[L7,W1]         EMIT T
[L8,W9/11] A3: JOIN 0
[L9,W8]         GOTO A0
```

```
 – Tick 1 –
!  reset ;
% In:
% Out: R S
% RT = 7
PAR_L1 PAR_L2 PARE_L3
EMIT_L4 EMIT_L5 PAUSE_L6
JOIN_L8
– Tick 2 –
% In:
% Out: R S T
PAUSE_L6 EMIT_L7 JOIN_L8
GOTO_L9
PAR_L1 PAR_L2 PARE_L3
EMIT_L4 EMIT_L5 PAUSE_L6
JOIN_L8
– Tick 3 –
% In:
% Out: R S T
PAUSE_L6 EMIT_L7 JOIN_L8
GOTO_L9
PAR_L1 PAR_L2 PARE_L3
EMIT_L4 EMIT_L5 PAUSE_L6
JOIN_L8
```

(a) Esterel (b) CKAG (c) KEP assembler (d) Sample trace

Fig. 2. A concurrent example program.

can be reduced to kernel statements. The KEP Instruction Set Architecture (ISA) includes all kernel statements, and in addition some frequently used derived statements. The KEP ISA also includes valued signals, which cannot be reduced to kernel statements. The only parts of Esterel v5 that are not part of the KEP ISA are combined signal handling and external task handling, as they both seem to be used only rarely in practice; however, adding these capabilities to the KEP ISA seems relatively straightforward.

Due to this direct mapping from Esterel to the KEP ISA, most Esterel statements can be executed in just one instruction cycle. For more complicated statements, well-known translations into kernel statements exist, allowing the KEP to execute arbitrary Esterel programs. Part of the KEP instruction set is shown in Figure 3. The KEP assembler programs corresponding to ExSeq and ExPar and sample traces are shown in Figures 1(c)/(d) and 2(c)/(d), respectively. Note that PAUSE is executed for at least two consecutive ticks, and consumes an instruction cycle at each tick.

The KEP provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body [15]. Therefore, no additional instruction cycles are needed to test for preemption. Only upon entering a preemption scope two cycles are needed to initialize the Watcher, as for example the $\text{WABORT}_{\text{L1}}$ instruction in ExSeq.

To implement concurrency, the KEP employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses and dynamically changing priorities. To begin of each instruction-cycle, the enabled thread with the highest priority is selected and executed. The scheduler is very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing the priority of a thread

Mnemonic, Operands	Esterel Syntax	Cycles	Notes
PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$ ... PAR *prio*$_n$, startAddr$_n$, *id*$_n$ PARE *endAddr* *startAddr*$_1$: ... *startAddr*$_2$: ... *startAddr*$_n$: ... *endAddr*: JOIN	[p$_1$ || ⋮ || p$_n$]	$\Big\} n+1$ 1	For each thread, one PAR is needed to define the start address, thread id and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is defined via PARE. The cycle count of a *fork node* depends on the count of threads.
PRIO *prio*		1	Set current thread priority to *prio*.
[W]ABORT[I, *n*] *S*, *endAddr* ... *endAddr*:	**[weak] abort** ... **when [immediate,** *n*] *S*	2	
SUSPEND[I,*n*] *S*, *endAddr* ... *endAddr*:	**suspend** ... **when [immediate,** *n*] *S*	2	
startAddr: ... EXIT *exitAddr startAddr* ... *exitAddr:*	**trap** *T* **in** ... **exit** *T* ... **end trap**	1	Exit from a trap, *startAddr*/*exitAddr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ||.
PAUSE	**pause**	1	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [I, *n*] *S*	**await [immediate,** *n*] *S*	1	
SIGNAL *S*	**signal** *S* **in** . . . **end**	1	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	1	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	**sustain** *S* [(*val*)]	1	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** . . . **end**	1	Jump to *elseAddr* if *S* is absent.
HALT	**halt**	1	Halt the program.
addr: . . . GOTO *addr*	**loop** . . . **end loop**	1	Jump to *addr*.

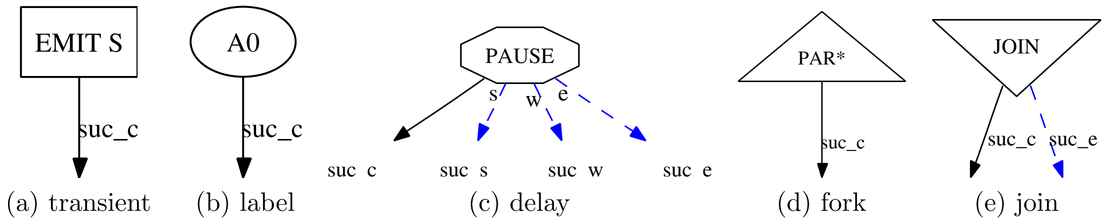
Fig. 3. Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

costs an instruction. For each thread, a PAR instruction is executed, to initialize the program counter and the priority and to define the thread id. Thereafter one PARE instruction is executed, which denotes the end of the parallel scope. During each instant in which one parallel thread is active, also the JOIN must be executed, in order to determine whether the threads have terminated.

3.3 The Concurrent Kep Assembler Graph (CKAG)

The WCRT analysis is not directly performed on the Esterel level, but on an intermediate data structure, the CKAG. The CKAG is a directed graph composed of various types of nodes and edges to match KEP program behavior. It is used during compilation from Esterel to KEP assembler, for, *e. g.*, dead code elimination, priority assigning [13], optimizations and the WCRT analysis.

The CKAG distinguishes *transient nodes*, which represent instantaneous execution, *delay nodes*, which represent statements that may hold for more than one



(a) transient (b) label (c) delay (d) fork (e) join

Fig. 4. Nodes and edges of a Concurrent KEP Assembler Graph (CKAG).

tick, and *fork* and *join nodes*, which represent concurrency (see Figure 4). Given a CKAG node n, the set $n.suc\_c$ denotes the set of sequential control flow successors (represented in the CKAG as solid edges). Successors reached via preemptions are $n.suc\_s$ for strong aborts, $n.suc\_w$ for weak aborts, and $n.suc\_e$ for exceptions (exit), represented as dashed edges; they are marked with small tail labels s, w and e, respectively. The CKAGs corresponding to ExSeq and ExPar can be found in Figures 1(b) and 2(b), respectively.

The CKAG is built from Esterel source by traversing recursively over its *Abstract Syntax Tree* (AST) generated by the *Columbia Esterel Compiler* (CEC) [8]. Visiting an Esterel statement results in creating the according CKAG node. A node typically contains exactly one statement, except *label nodes* containing just address labels and *fork nodes* containing one PAR statement for each child thread initialization and a PARE statement. When a *delay node* is created, additional preemption edges are added according to the abortion/exception context.

To preserve the signal-dependencies in the execution, additional priority assignments (PRIO statements) might be introduced by the compiler. To assure schedulability, the program is completely dismantled, *i. e.*, transformed into kernel statements. In this dismantled graph the priority assignments are inserted. A subsequent “undismantling” step before the computation of the WCRT detects specific patterns in the CKAG and collapses them to more complex instructions, such as AWAIT or SUSTAIN, which are also part of the KEP instruction set.

4 Worst Case Reaction Time (WCRT)

Given a KEP program we define its WCRT as the maximum number of KEP cycles executable in one instant. Thus WCRT analysis requires finding the longest instantaneous path in the CKAG, where the length metric is the number of required KEP instruction cycles. We abstract from signal relationships and might therefore consider unfeasible executions. Therefore the computed WCRT can be pessimistic. We first present, in Section 4.1, a restricted form of the WCRT algorithm that does not handle concurrency yet. The general algorithm requires an analysis of instant reachability between *fork* and *join* nodes, which is discussed in Section 4.2, followed by the presentation of the general WCRT algorithm in Section 4.3.

4.1 Sequential WCRT Algorithm

First we present a WCRT analysis of sequential CKAGs (no *fork* and *join nodes*). Consider the ExSeq example in Figure 1(a) again. The longest possible execution

```
int getWcrtSeq($g$)    // Compute WCRT for sequential CKAG $g$
  forall $n \in Nodes$ do $n.inst := n.next := \perp$ end
  getInstSeq($g.root$)
  forall $d \in DelayNodes$ do getNextSeq($d$) end
  return max ($\{g.root.inst\} \bigcup \{d.next : d \in DelayNodes\}$)
end
```

```
int getInstSeq($n$)    // Compute statements instantaneously reachable from node $n$
  if $n.inst = \perp$ then
    if $n \in TransientNodes \cup LabelNodes$ then
      $n.inst :=$ max $\{$getInstSeq$(c) : c \in n.suc\_c\}$ + cycles($n.stmt$)
    elif $n \in DelayNodes$ then
      $n.inst :=$ max $\{$getInstSeq$(c) : c \in n.suc\_w \cup n.suc\_e\}$ + cycles($n.stmt$)
    fi
  fi
  return $n.inst$
end
```

```
int getNextSeq($d$)    // Compute statements instantaneously reachable from delay node $d$ at tick start
  if $d.next = \perp$ then
    $d.next :=$ max $\{$getInstSeq$(c) : c \in d.suc\_c \cup d.suc\_s\}$ + cycles($d.stmt$)
  fi
  return $d.next$
end
```

Fig. 5. WCRT algorithm, restricted to sequential programs. The nodes of a CKAG g are given by $Nodes = TransientNodes \cup LabelNodes \cup DelayNodes \cup ForkNodes \cup JoinNodes$, $g.root$ indicates the first KEP statement. cycles($stmt$) returns the number of instruction cycles to execute $stmt$, see third column in Figure 3.

occurs when the signal I becomes present, as is the case in Tick 3 of the example trace shown in Figure 1(d). Since the abortion triggered by I is weak, the abort body is still executed in this instant, which takes four instructions: $\mathsf{PAUSE}_{\mathsf{L2}}$, $\mathsf{EMIT}_{\mathsf{L3}}$, the $\mathsf{GOTO}_{\mathsf{L4}}$, and $\mathsf{PAUSE}_{\mathsf{L2}}$ again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and $\mathsf{EMIT}_{\mathsf{L5}}$ and $\mathsf{HALT}_{\mathsf{L6}}$ are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via a *Depth First Search* (DFS) traversal of the CKAG, see the algorithm in Figure 5. For each node n a value $n.inst$ is computed, which gives the WCRT from this node on in the same instant when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For non-instantaneous *delay nodes* we distinguish two cases within a tick: control can *reach* a *delay node* d, meaning that the thread executing d has already executed some other instructions in that tick, or control can *start* in d, meaning that d must have been reached in some preceding tick. In the first case, the WCRT from d on within an instant is expressed by the $d.inst$ variable already introduced. For the second case, an additional value $d.next$ stores the WCRT from d on within an instant; "next" here expresses that in the CKAG traversal done to analyze the overall WCRT, the $d.next$ value should not be included in the current tick, but in a next tick. Having these two values ensures that the algorithm terminates in the case of non-instantaneous loops: to compute $d.next$ we might need the value $d.inst$.

For a *delay node*, we also have to take abortions into account. The handlers (*i. e.*, their continuations—typically the end of an associated abort/trap scope) of weak abortions and exceptions are instantaneously reachable, so their WCRTs are added to the $d.inst$ value. In contrast, the handlers of strong abortions cannot be executed in the same instant the delay node is reached, because according to the Esterel semantics an abortion body is not executed at all when the abortion takes place. On the KEP, when a strong abort takes place, the delay nodes where the control of the (still active) threads in the abortion body resides are executed once, and then control moves to the abortion handler. In other words, control cannot move from a delay node d to a (strong) abortion handler when control reaches d, but only when it starts in d. Therefore, the WCRT of the handler of a strong abortion is added to $d.next$, and not to $d.inst$.

We do not need to take a weak abortion into account for $d.next$, because it cannot contribute to a longest path. An abortion in an instant when a delay node is reached will always lead to a higher WCRT than an execution in a subsequent instant where a thread starts executing in the delay node.

The resulting WCRT for the whole program is computed as the maximum over all WCRTs of nodes where the execution may start. These are the *start node* and all *delay nodes*. To take into account that execution might start simultaneously in different concurrent threads, we also have to consider the $next$ value of *join nodes*.

Consider the example ExSeq in Figure 1. Each node n in the CKAG g is annotated with a label "$\mathsf{W}\langle n.inst\rangle$" or, for a delay node, a label "$\mathsf{W}\langle n.inst\rangle/\langle n.next\rangle$." In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers $\mathsf{L}\langle n\rangle$. It is $g.root = \mathsf{L1}$. The sequential WCRT computation starts initializing the $inst$ and $next$ values of all nodes to $\perp$ (line 2 in getWcrtSeq, Figure 5). Then getInstSeq(L1) is called, which computes $\mathsf{L1}.inst := \mathsf{max}\ \{\ \mathsf{getInstSeq(L2)}\ \} + \mathsf{cycles}(\mathsf{WABORT}_{\mathsf{L1}})$. The call to getInstSeq(L2) computes and returns $\mathsf{L2}.inst := \mathsf{cycles}(\mathsf{PAUSE}_{\mathsf{L2}}) + \mathsf{cycles}(\mathsf{EMIT}_{\mathsf{L5}}) + \mathsf{cycles}(\mathsf{HALT}_{\mathsf{L6}}) = 3$, hence $\mathsf{L1}.inst := 3 + 2 = 5$. Next, in line 4 of getWcrtSeq, we call getNextSeq(L2), which computes $\mathsf{L2}.next := \mathsf{getInstSeq(L3)} + \mathsf{cycles}(\mathsf{PAUSE}_{\mathsf{L2}})$. The call to getInstSeq(L3) computes and returns $\mathsf{L3}.inst := \mathsf{cycles}(\mathsf{EMIT}_{\mathsf{L3}}) + \mathsf{cycles}(\mathsf{GOTO}_{\mathsf{L4}}) + \mathsf{L2}.inst = 1 + 1 + 3 = 5$. Hence $\mathsf{L2}.next := 5 + 1 = 6$, which corresponds to the longest path triggered by the presence of signal I, as we have seen earlier. The WCRT analysis therefore inserts an "EMIT _TICKLEN, #6" instruction before the body of the KEP assembler program to initialize the TickManager accordingly.

4.2 Instantaneous Statement Reachability for Concurrent Esterel Programs

It is important for the WCRT analysis whether a *join* and its corresponding *fork* can be executed within the same instant. The algorithm for instantaneous statement reachability computes for a *source* and a *target* node whether the *target* is reachable instantaneously from the *source*. Source and target have to be in sequence to each other, *i. e.*, not concurrent, to get correct results.

In simple cases like EMIT or PAUSE the sequential control flow successor is executed in the same instant respectively next instant, but in general the behavior is more complicated. The parallel, *e. g.*, will terminate instantaneously if all subthreads are instantaneous or an EXIT will be reached instantaneously; it is not-instantaneous if at least one sub-thread is not instantaneous.

The complete algorithm is presented in detail elsewhere [4]. The basic idea is to compute for each node three potential reachability properties: *instantaneous*, *not-instantaneous*, *exit-instantaneous*. Note that a node might be as well (potentially) *instantaneous* as (potentially) *non-instantaneous*, depending on the signal status. Computation begins by setting the *instantaneous* predicate of the *source* node to *true* and the properties of all other nodes to *false*. When any property is changed, the new value is propagated to its successors. If we have set one of the properties to *true*, we will not set it to *false* again. Hence the algorithm is monotonic and will terminate. Its complexity is determined by the amount of property changes which are bounded to three (three boolean) for all nodes, so the complexity is $O(3 * |Nodes|) = O(|Nodes|)$.

The most complicated computation is the property *instantaneous* of a *join node* because several attributes have to be fulfilled for it to be *instantaneous*:

- For each thread, there has to be a (potentially) instantaneous path to the *join node*.
- The predecessor of the *join node* must not be an EXIT, because EXIT nodes are no real control flow predecessors. At the Esterel level, an exception (exit) causes control to jump directly to the corresponding exception handler (at the end of the corresponding trap scope); this jump may also cross thread boundaries, in which case the threads that are jumped out of and their sibling threads threads terminate. To emulate this at the KEP level, an EXIT instruction does not jump directly to the exception handler, but first executes the JOIN instructions on the way, to give them the opportunity to terminate threads correctly. If a JOIN is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler, or to another intermediate JOIN. To express this, we use the third property besides *instantaneous* and *non-instantaneous*: *exit-instantaneous*.

Roughly speaking the *instantaneous* property is propagated via for-all quantifier, *not_instantaneous* and *exit_instantaneous* via existence-quantifier.

Most other nodes simply propagate their own properties to their successors. The *delay node* propagates in addition its *non-instantaneous* predicate to its delayed successors and *exit nodes* propagate *exit-instantaneous* reachability, when they themselves are reachable instantaneously .

4.3 General WCRT Algorithm

The general algorithm, which can also handle concurrency, is shown in Figure 6. It emerges from the sequential algorithm that has been described in Section 4.1 by enhancing it with the ability to compute the WCRT of *fork* and *join nodes*. Note that the instantaneous WCRT of a *join node* is needed only by a *fork node*, all *transient nodes* and *delay nodes* do not use this value for their WCRT. The WCRT of the *join node* has to be accounted for just once in the instantaneous WCRT of its corresponding *fork node*, which allows the use of a DFS-like algorithm.

The instantaneous WCRT of a *fork node* is simply the sum of the instantaneously reachable statements of its sub-threads, plus the PAR statement for each sub-thread and the additional PARE statement.

```
int getWcrt($g$)     // Compute WCRT for a CKAG $g$
  forall $n \in Nodes$ do $n.inst := n.next := \bot$ end
  forall $d \in DelayNodes$ do getNext($d$) end
  forall $j \in JoinNodes$ do getNext($j$) end // Visit according to hierarchy (inside out)
  return max ({getInst($g.root$)} $\bigcup$ {$n.next : n \in DelayNodes \cup JoinNodes$})
end
```

```
int getInst ($n$)      // Compute statements instantaneously reachable from node $n$
  if $n.inst = \bot$ then
    if $n \in TransientNodes \cup LabelNodes$ then
      $t.inst$ := max {getInst($c$) : $c \in suc\_c \setminus JoinNodes$} + cycles($n.stmt$)
    elif $n \in DelayNodes$ then
      $n.inst$ := max {getInst($c$) : $c \in suc\_w \cup suc\_e \setminus JoinNodes$} + cycles($n.stmt$)
    elif $n \in ForkNodes$ then
      $n.inst := \sum_{t \in n.suc\_c} t.inst$ + cycles($n.par\_stmts$) + cycles(PARE)
      $prop$ := reachability($n, n.join$) // Compute instantaneous reachability of join from fork
      if $prop.instantaneous$ or $prop.exit\_instantaneous$ then
        $n.inst$ += getInst($n.join$)
      elif $prop.not\_instantaneous$ then
        $n.inst$ += cycles(JOIN) // JOIN is always executed
      fi
    elif $n \in JoinNodes$ then
      $n.inst$ := max{getInst($c$) : $c \in suc\_c \cup suc\_e$} + cycles($n.stmt$);
    fi
  fi
  return $n.inst$
end
```

```
int getNext($n$)      // Compute statements instantaneously reachable from delay node $d$ at tick start
  if $n.next = \bot$ then
    if $n \in DelayNodes$ then
      $n.next$ := max {getInst($c$) : $c \in suc\_c \cup suc\_s \setminus JoinNodes$} + cycles($n.stmt$)
    elif $n \in JoinNodes$ and $(n.fork, n).not\_instantaneous$ then
      $n.next := n.inst + \sum_{t \in n.fork.suc\_c} max\{n.next : t.id = n.id\}$
    fi
  fi
  return $n.next$
end
```

Fig. 6. General WCRT algorithm.

Like *delay nodes*, *join nodes* also have a *next* value. When a *fork/join* pair (f, j) could be *non-instantaneous* then we have to compute a WCRT *j.next* for the next instants analogously to the *delay nodes*. Its computation requires first the computation of all sub-thread *next* WCRTs. Then we simply sum the maximum value for every thread. If the parallel does not terminate instantaneously, all directly reachable states are reachable in the next instant. Therefore we have to add the execution time for all statements that are instantaneously reachable from the *join node*. Note that the computation is independent from the scheduling.

The whole algorithm computes first the *next* WCRT for all *delay* and *join nodes*; it computes recursively all needed *inst* values. Thereafter the instantaneous WCRT for all remaining nodes is computed. The result is simply the maximum over all computed values.

Consider the example in Figure 2. First we note that the *fork/join* pair is always *non-instantaneous*, due to the cycles($\mathsf{PAUSE_{L6}}$) statement. We compute L6.$next$ = cycles($\mathsf{PAUSE_{L6}}$) + cycles($\mathsf{EMIT_{L7}}$) $= 2$. From the *fork node* L3, the PAR and PARE statements, the instantaneous parts of both threads and the JOIN are executed, hence $\mathsf{L3}.inst = 2 \times \mathsf{cycles(PAR)} + \mathsf{cycles(PARE)} + \mathsf{cycles(JOIN)} + \mathsf{L4}.inst + \mathsf{L5}.inst = 2+1+1+1+2 = 7$. Therefore, the WCRT of the program is $\mathsf{L8}.next = \mathsf{L6}.next + \mathsf{L8}.inst = 2 + 9 = 11$. Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel-programs is that due to nesting of exceptions and concurrency, statements might be executed multiple times in one instant. This problem, also known as *reincarnation*, is handled correctly by our algorithm. Since we compute nested joins from inside to outside, the same statement may effect both the instantaneous and non-instantaneous WCRT, which are added up in the next join. This exactly matches the possible control-flow in case of reincarnation. Even when a statement is executed multiple times in an instant, we compute a correct upper bound for the WCRT.

Regarding the complexity of the algorithm, let $n := |Nodes|$, $d := |DelayNodes|$, $f := |ForkNodes|$ and $j := |JoinNodes|$. For each node its WCRT's *inst* and *next* are computed at most once, and for all *fork nodes* a fork-join reachability analysis is additionally made, which has itself $O(n)$. So we get altogether a complexity of $O(n + d + j) + O(f * n) = O(2 * n) + O(n^2) = O(n^2)$.

5 Experimental Results

The WCRT analysis is implemented in the KEP compiler. It automatically inserts a correct EMIT _TICKLEN instruction at the beginning of the program. To validate our approach, we used Esterel-Studio to generate test cases for Esterel programs, which cover all states and transitions. The programs were executed on the KEP with the test cases as input. We measured the maximal reaction time during these executions and compared it to the computed value. The Esterel programs in Table 1 are taken from the *Estbench* [6]. We never underestimated the WCRT, and our results are on average 38% too high. For each program, the lines of code, the computed WCRT and the measured WCRT with the resulting difference is given. We also give the average WCRT analysis time on a standard PC (AMD Athlon XP, 2.2GHz, 512 KB Cache, 1GB Main Memory); as the table indicates, the analysis takes only a couple of milliseconds.

The table also compares the Average Case Reaction Time (ACRT) with the WCRT. The ACRT is on average about two thirds of the WCRT, which is relatively high compared to traditional architectures. In other words, the worst case on the KEP is not much worse than the average case, and padding the tick length according to the WCRT does not waste too much resources. On the same token, designing for worst-case performance, as typically must be done for hard real-time systems, does not cause too much overhead compared to the typical average-case performance design. Finally, the table also lists the number of scenarios generated by Esterel-Studio and accumulated logical tick count for the test traces.

6 Conclusions and Further Work

We have presented the WCRT analysis of reactive programs written in the Esterel language. The analysis is performed on a graph representation, the *Concurrent KEP Assembler Graph* (CKAG). In a first step we compute whether concurrent threads

Module name	LoC	WCRT			t_{an}	ACRT		Test	Ticks
		WC_{est}	WC_{act}	$r_{est-act}$	[ms]	AC_{act}	AC/WC	cases	
abcd	152	56	44	27%	1.0	28	64%	161	673
abcdef	232	84	68	24%	1.5	42	62%	1457	50938
eight_buttons	312	112	92	22%	2.0	57	62%	13121	45876
channel_protocol	57	49	38	29%	0.4	19	50%	114	556
reactor_control	24	24	15	60%	0.2	12	80%	6	20
runner	26	10	7	43%	0.3	4	57%	131	2548
tcint	410	192	138	39%	2.8	86	62%	148	1325

Table 1
Experimental results. The WC_{est} and WC_{act} data denote the estimated and actual WCRT, respectively, measured in instruction cycles. The ratio $r_{est-act} := WC_{est}/WC_{act} - 1$ indicates by how much our analysis overestimates the WCRT. AC_{act} is the actual Average Case Reaction Time (ACRT), AC/WC ($= AC_{act}/WC_{act}$) gives the ratio to the WCRT. Test cases and Ticks are the number of different scenarios and logical ticks that were executed, respectively.

terminate instantaneously, thereafter we are able to compute for each statement how many instruction are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program. The analysis considers concurrency and the multiple forms of preemption that Esterel offers. The asymptotic complexity of the WCRT analysis algorithm is quadratic in the size of the program; however, experimental results indicate that the overhead of WCRT analysis as part of compilation is negligible. We have implemented this analysis as part of a compiler from Esterel to KEP assembler, and use it to automatically compute an initialization value for the KEP's TickManager. This allows to achieve a high, constant response frequency to the environment, and can also be used to detect hardware errors by detecting timing overruns.

Our analysis is safe, *i. e.*, conservative in that it never underestimates the WCRT, and it does not require any user annotations to the program. In our benchmarks it overestimates the WCRT on average by about 40%. This is already competitive with the state of the art in general WCET analysis, and we expect this to be acceptable in most cases. However, there is still significant room for improvement. So far, we are not taking any signal status into account, therefore our analysis includes some unreachable paths. Considering all signals would lead to an exponential growth of the complexity, but some local knowledge should be enough to rule out most unreachable paths of this kind. Also a finer grained analysis of which parts of parallel threads can be executed in the same instant could lead to better results. However, it is not obvious how to do this efficiently.

Our analysis is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. At least for non-parallel programs our approach should be of value for other compilation methods for Esterel as well, *e. g.*, simulation-based code generation. A virtual machine with similar support for concurrency could also benefit from our approach. We would also like to generalize our approach to handle different ways to implement concurrency. A WCRT analysis directly on the Esterel level gives information on the longest possible execution path. Together with a known translation to C, this WCRT information could be combined with a traditional WCET analysis, which takes caches and other hardware details into account.

References

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

[2] Berry, G. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999.

[3] Berry, G. *The Esterel v5 Language Primer, Version v5_91.* Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[4] Boldt, M. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Nov. 2007. To appear.

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

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

[7] Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In *Electronic Notes in Theoretical Computer Science* (July 2002), F. Maraninchi, A. Girault, and E. Rutten, Eds., vol. 65, Elsevier.

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

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

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

[11] Edwards, S. A., Kapadia, V., and Halas, M. Compiling Esterel into static discrete-event code. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'04)* (Barcelona, Spain, Mar. 2004).

[12] Harel, D., and Pnueli, A. On the development of reactive systems. *Logics and models of concurrent systems* (1985), 477–498.

[13] Li, X., Boldt, M., and von Hanxleden, R. Mapping Esterel onto a multithreaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, October 21–25 2006).

[14] Li, X., Lukoschus, J., Boldt, M., Harder, M., and von Hanxleden, R. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (New York, NY, USA, Sept. 2005), ACM Press, pp. 225–236.

[15] Li, X., and von Hanxleden, R. 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).

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

[17] Logothetis, G., Schneider, K., and Metzler, C. 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.

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

[19] Potop-Butucaru, D., and de Simone, R. *Optimization for faster execution of Esterel programs.* Kluwer Academic Publishers, Norwell, MA, USA, 2004, pp. 285–315.

[20] Puschner, P., and Burns, A. A review of worst-case execution-time analysis (editorial). *Real-Time Systems 18*, 2/3 (2000), 115–128.

[21] Ringler, T. Static worst-case execution time analysis of synchronous programs. In *ADA-Europe- 5. International Conference on Reliable Software Technologies* (2000).

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

[23] Tardieu, O., and de Simone, R. Instantaneous termination in pure Esterel. In *Static Analysis Symposium* (San Diego, California, June 2003).