Worst Case Reaction Time Analysis for a Synchronous Concurrent Processor

Marian Boldt

2007-07-02

Department of Computer Science
Real-Time and Embedded Systems Group

Advised by:
Reinhard von Hanxleden
Claus Traulsen
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

__________________________
Abstract

Reactive programs have to react continuously to their inputs with according output. 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 thesis, I 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, I 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.

I 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 25% on average.
## Contents

1. **Introduction**  
   1.1. Esterel .......................... 3  
   1.2. The Kiel Esterel Processor (KEP) .................. 3  
   1.3. The Concurrent KEP Assembler Graph (CKAG) .............. 5  
   1.4. Related Work ....................... 7  

2. **Instantaneous Paths**  
   2.1. Node Successor Definitions .................. 9  
   2.2. Sequential Paths .......................... 10  
   2.3. Parallel Paths ............................ 11  
   2.4. Fork-Join Reachability .................... 12  
   2.5. General Statement Reachability ................. 12  

3. **Worst Case Reaction Time Analysis**  
   3.1. KEP Instruction Cycles .................... 15  
   3.2. Sequential WCRT Analysis .................. 16  
   3.3. General WCRT Analysis .................... 17  
   3.4. WCRT Overestimation ...................... 20  
   3.5. Experimental Results ..................... 21  

4. **Implementation**  
   4.1. class WCRT ............................ 27  
   4.2. class WCRT_Sequential .................... 28  
   4.3. class WCRT_Parallel ...................... 28  
   4.4. Usage ................................ 29  

5. **Conclusions and Further Work** ................. 31  

6. **Bibliography** ................................ 33  

7. **Appendix** ................................ 37  
   A. C++ Sources ............................... 37
List of Figures

1.1. Overview of the Esterel syntax and how these Esterel statements are compiled to the KEP instruction set. Cycles give the number of processor cycles needed for the execution. .................................................. 4

1.2. Nodes and edges of a Concurrent KEP Assembler Graph (CKAG). ......................... 6

1.3. CKAG Building (Esterel source, KEP assembler, CKAG) ..................................... 7

2.1. A sequential Esterel example (a), the generated KEP assembler (b), the corresponding CKAG (c), and its traces (d). The body of the KEP assembler program (without interface declaration) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. ............................................................... 11

2.2. An Esterel example with concurrency (a), the resulting KEP assembler program (b) and CKAG (c) to show different kinds of instantaneous parallel paths (d). These paths are not extensible and reachable: signal A has to be present and absent respectively to reach path 1a and 1b respectively. So these paths are traces. ........................................... 14

3.1. The sequential Esterel example shown in Figure 2.1 (a), the corresponding CKAG annotated with the results of its WCRT analysis (b), the generated KEP completed with the initialization of the TickManager (c), and a sample execution trace (d). The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles. ......................................................... 18

3.2. WCRT algorithm, restricted to sequential programs. Function return values may be ignored. The nodes of a CKAG g are given by $\text{Nodes} = \text{TransientNodes} \cup \text{LabelNodes} \cup \text{DelayNodes} \cup \text{ForkNodes} \cup \text{JoinNodes}$, $g.root$ indicates the first KEP statement. $\text{cycles}(\text{stmt})$ returns the number of instruction cycles to execute stmt, see third column in Figure 1.1]. ......................................................... 19

3.3. A concurrent example program. ............................................................................. 20

3.4. General WCRT algorithm. ...................................................................................... 23

3.5. The standard \textit{ABRO} example: there are strong abort edges between the delay nodes with \textit{AWAIT} statements to a node outside the parallel indicated by a different thread-id. So their next WCRT values are used for next value of the join node. ......................................................... 24
List of Figures

3.6. Inconsistent sequential path example: signal $I$ cannot be present and absent within the same tick due to Esterel semantics, so this program has two inconsistent paths. ........................................... 24

3.7. Inconsistent parallel path example ........................................... 25

3.8. Unreachable parallel path example: a more exact reachability analysis would be necessary to determine whether statements of different threads are executable within the same instant. .............................. 25

4.1. The visitor pattern: the conditional in (a) is replaced by calling the welcome method (b). The conditional bodies $body_i$ are implemented by a specific visit method which is called by welcome. ............................. 29
1. Introduction

Many embedded and hard real-time systems belong to the class of reactive systems, which continuously react to inputs from the environment by generating corresponding outputs. Therefore exact timing information or at least an upper bound of the execution time is crucial for these systems. To perform an exact Worst Case Execution Time (WCET) analysis is difficult, and not possible in general 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 \[21, 5\]. 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 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 \[23, 17, 18\]. These processors directly support reactive control flow, such as preemption and concurrency. In this thesis I will use the Kiel Esterel Processor (KEP), a reactive processor based on the synchronous language Esterel, to show that timing analysis is practical for
1. Introduction

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 analysis and optimizations that go beyond WCRT analysis.

The WCRT analysis is performed during the compilation of an Esterel program to KEP assembler on a graph structure, called Concurrent KEP Assembler Graph (CKAG), which represents the resulting KEP control flow. The remainder of this chapter will give a short overview of Esterel, the KEP and KEP assembler, the CKAG and related work. In Chapter 2 we consider paths in particular instantaneous paths of an Esterel program. Chapter 3 explains the algorithm to determine the WCRT, while Chapter 4 gives details on the implementation. The results and open problems are discussed in Chapter 5.

Parts of this work were presented during the Proceedings of the Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P’07) in Braga, Portugal, 2007 [4].
1.1. Esterel

Classical programming languages like C/C++ are designed to handle data from standard keyboard or file input/output to compute some result. The result has to be correct and the computation should not take too much time. How many milliseconds are needed in detail is not relevant for the result. Not so in the real-time and embedded systems world, these the correctness of results depends on timing behavior and the programs has to react withing specific timings. Implementing these features in languages with a standard data-oriented programming paradigm is very difficult, because a time model must be implemented by hand. Furthermore preemption and concurrency are hard to express. The reactive and synchronous languages, like Signal [14], Lustre [15] and Esterel [3], support these real-time demands directly in syntax and semantics.

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. Most statements are transient including 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 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 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 complications when traps are nested or when the trap scope includes concurrent threads. The following rules apply: 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. All Esterel statements can be translated into a small set of kernel statements.

1.2. The Kiel Esterel Processor (KEP)

Synchronous reactive and preemptive behavior is not common to standard hardware and must be implemented in software, which is usually inefficient. To solve this problem the KEP is designed to implement Esterel directly in hardware, whereby
## 1. Introduction

<table>
<thead>
<tr>
<th>Esterel Source</th>
<th>KEP Assembler</th>
<th>Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>input ( I ) [: type]</td>
<td>( \text{INPUT}[V] ) ( I )</td>
<td>0</td>
<td>type ( \in { \text{integer, boolean} } ) ( I/O)-statements are part of the interface: no instruction cycles are needed.</td>
</tr>
<tr>
<td>output ( O ) [: type]</td>
<td>( \text{OUTPUT}[V] ) ( O )</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>emit ( S ) [: {val}]</td>
<td>( \text{EMIT} ) ( S \cdot { \text{#data</td>
<td>reg} } )</td>
<td>1</td>
</tr>
<tr>
<td>present ( S ) then ... else ... end present</td>
<td>( \text{PRESEN} ) ( S \cdot { \text{elseAddr} } )</td>
<td>1</td>
<td>Jump to ( \text{elseAddr} ) if ( S ) is absent. No ( \text{GOTO} ) statement when the else body is empty.</td>
</tr>
<tr>
<td>[\text{weak}] abort ... when [immediate, ( n ) ( S )</td>
<td>( \text{LOAD _COUNT} \cdot \text{[n]} ) ( \text{W} \text{ABORT} \cdot { \text{endAddr} } )</td>
<td>1</td>
<td>To delay a preemption for ( n ) ticks is done by setting the built-in variable ( \text{_COUNT} ).</td>
</tr>
<tr>
<td>suspend ... when [immediate, ( n ) ( S )</td>
<td>( \text{SUSPEND} \cdot { \text{endAddr} } )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>trap ( T ) in ... exit ( T ) ... end trap</td>
<td>startAddr: ... ( \text{EXIT} ) exitAddr startAddr ... exitAddr:</td>
<td>1</td>
<td>Exit from a trap, startAddr/exitAddr specifies trap scope. Unlike ( \text{GOTO} ), check for concurrent ( \text{EXITs} ) and terminate enclosing ( \text{1} ).</td>
</tr>
<tr>
<td>pause</td>
<td>( \text{PAUSE} )</td>
<td>1</td>
<td>Wait for a signal. ( \text{AWAIT TICK} ) is equivalent to ( \text{PAUSE} ).</td>
</tr>
<tr>
<td>await [immediate, ( n ) ( S )</td>
<td>( \text{AWAIT} \cdot { l \cdot n } ) ( S )</td>
<td>1</td>
<td>Initialize a local signal ( S ).</td>
</tr>
<tr>
<td>signal ( S ) in ... end</td>
<td>( \text{SIGNAL} \cdot { \text{endAddr} } )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sustain ( S ) [: {val}]</td>
<td>( \text{SUSTAIN} \cdot { \text{endAddr} } )</td>
<td>1</td>
<td>Sustain (valued) signal ( S ).</td>
</tr>
<tr>
<td>halt</td>
<td>( \text{HALT} )</td>
<td>1</td>
<td>Halt the program.</td>
</tr>
<tr>
<td>loop ... end loop</td>
<td>addr: ... ( \text{GOTO} ) addr</td>
<td>1</td>
<td>Jump to addr. loop body has to be noninstantaneous</td>
</tr>
<tr>
<td>( [\begin{align*} p_1 \end{align*}] )</td>
<td>( \text{PAR} ) ( \text{prio}_1, \text{startAddr}_1, \text{id}_1 ) ... ( \text{PAR} ) ( \text{prio}_n, \text{startAddr}_n, \text{id}_n ) ( \text{PARE} ) ( \text{endAddr} ) ... ( \text{startAddr}_2 ) ... ( \text{startAddr}_n ) ... ( \text{endAddr} ) ( \text{JOIN} )</td>
<td>( n + 1 )</td>
<td>For each thread, one ( \text{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 ( \text{PARE} ). The cycle count of a fork node depends on the count of threads. Behind ( \text{endAddr} ) the corresponding join node occurs which is executed at the end of each instant the parallel is active, consuming one instruction cycle. A ( \text{JOIN} ) statement is executed a second time when it is part of a nested parallel.</td>
</tr>
<tr>
<td>PRIO ( \text{prio} )</td>
<td>1</td>
<td>Current thread priority is set to ( \text{prio} ). This statement has no Esterel counterpart, it is implementing the Esterel semantics.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1.: Overview of the Esterel syntax and how these Esterel statements are compiled to the KEP instruction set. Cycles give the number of processor cycles needed for the execution.
the according assembler instruction set is based on the Esterel language [3].

Many Esterel statements can be found directly in the KEP syntax, while more complex statements have to be dismantled into simpler statements. In particular, all kernel statements and some frequently used derived statements can be directly mapped to single KEP instructions. Hence 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 1.1. The KEP assembler programs corresponding to ExSeq and ExPar and sample traces are shown in Figures 3.1(c)/(d) and 3.3(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 is in the corresponding preemption body [18]. 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 WABORT_L1 instruction in ExSeq.

To implement concurrency, the KEP employs a multi-threaded architecture, where each thread has an independent program counter and threads are scheduled according to their statuses, thread-id and dynamically changing priorities: between all active threads, the thread with the highest priority is scheduled, if there is more than one thread, the highest thread-id counts. The scheduler is light-weight: scheduling and context switching do not cost extra instruction cycles, only changing the priority of a thread costs an instruction.

To initialize \( n \) parallel threads, \( n \) PAR instructions are executed, which initialize the program counter and the priority for each thread. Thereafter one PARE instruction is executed, which stores the end of the parallel scope. During each instant in which at least one parallel thread is active, also the according JOIN statement must be executed in order to determine whether the threads have terminated.

1.3. The Concurrent KEP Assembler Graph (CKAG)

The Concurrent KEP Assembler Graph (CKAG) is an intermediate data structure used for compilation from Esterel to KEP assembler. It represents KEP control flow behavior to perform complex computations like priority assigning [16], dead code elimination, statement collapsing and especially the Worst Case Reaction Time (WCRT) analysis.

1.3.1. Node and Edge Types

The CKAG distinguishes transient nodes, which represent instantaneous execution, delay nodes, which represent statements that may hold for more than one tick, and fork and join nodes, which represent concurrency (see Figure 1.2). Given a CKAG node \( n \), the set \( n.suc_c \) denotes the set of sequential control flow successors and are represented as solid edges in the CKAG. Successors reached via preemptions are
1. Introduction

Figure 1.2.: Nodes and edges of a Concurrent KEP Assembler Graph (CKAG).

...for strong aborts, $n.suc_w$ for weak aborts, and $n.suc_e$ for exit exceptions. The edges are represented as dashed edges, marked with small tail labels $s$, $w$ and $e$, respectively. Note that according to the semantics of Esterel preemption edges occur only from delay nodes, because a preemption either takes place at the beginning or the end of an execution, when a pause is reached.

1.3.2. Graph Building

The CKAG is built from the Esterel source by traversing recursively over its Abstract Syntax Tree (AST) generated by the Columbia Esterel Compiler (CEC) [8].

While the Esterel statements are compiled to KEP Assembler (KASM) statements, the corresponding CKAG is built by creating a node for each statement, which will be inserted into the graph. The kind of the node depends on the kind of the statement: for instantaneously executed statements a transient node, for address labels a label node, for non-instantaneous executed statements a delay node and for concurrency a fork node respectively join node.

A node typically contains exactly one statement, except label nodes containing only 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, e.g. in Figure 1.3 (c) a preemption edge from a PAUSE delay node to a label node of a strong abort is added because the delay node is to be within the body of this strong abort.

The delay nodes contain mostly PAUSE statements, since more complex potentially non-instantaneous statements like AWAIT, SUSTAIN and HALT are -at least initially- dismantled into kernel statements.

Some simple CKAG building examples respectively schemes are shown in Figure 1.3 together with the corresponding KEP program and the Esterel source they are built from. The first example (a) explains how an Esterel sequence is transformed, it remains in each case as a sequence: the KEP program is a KEP statement sequence and the CKAG is a node sequence. The next example (b) shows how an Esterel loop is handled. The loop body is translated recursively ahead of the loop a start address A0 and at the end the corresponding GOTO A0 statement are inserted. In the CKAG the behavior of restarting the loop body is reflected by the
edge between GOTO node and address label. In Figure 1.3 (c) the abort also handles recursively a body by adding preemption edges from all occurring delay nodes to the abort end. This is done for the preemption statements abort, weak abort and trap. For all of these statements a preemption edge with the according type and symbol is added. In this example at least one edge for each delay node is added with type \( s \) and symbol \( A \). Figure 1.3 (d) depicts the use of KEP concurrency. If two threads are defined to be in parallel by Esterel, then for each of them a thread-id and a start address is assigned via PAR statements. Each thread gets the initial priority 1 during creation, which might be changed during priority assigning [16]. Determining the end of the whole parallel the PARE defines an end address. Within the CKAG the PAR and the PARE statements are encapsulated by their fork node.

1.4. Related Work

As mentioned in the introduction, there exist numerous approaches to classical WCET analysis. For a survey see, e.g., Puschner and Burns [23]. These approaches usually consider (subsets) of general purpose languages, such as C, and take information on the processor designs and caches into account.

Regarding the analysis of synchronous programs, Logothetis, Schneider and Met-
1. Introduction

Zler [20, 19] 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. 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 [26]. 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.

Apart from 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, 22, 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 [24] considers the WCET analysis of C code generated from Esterel. However, 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. [17] 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 [17], 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. [17] for details on the relation between the maximum number of instruction per logical tick and the timing constraints from the environment perspective.
2. Instantaneous Paths

The goal of the WCRT analysis is to find the longest executable execution trace per instant measured in KEP instruction cycles. We identify execution traces of a KEP program $p$ as instantaneous paths in the according CKAG $C(p)$. Based on sequential paths the more general parallel paths will be defined.

2.1. Node Successor Definitions

A CKAG $C$ for a KEP program $p$ has the form

$$C = C(p) = (V, E, P)$$

with node set $V$, edge set $E \subseteq V \times V$ and preemption edges $P$.

$V$ is partitioned into three different types of nodes in the case of sequential programs and five types for parallel programs, respectively:

$$V = \text{TransientNodes}(C) \cup \text{LabelNodes}(C) \cup \text{DelayNodes}(C)$$

$$\cup \text{ForkNodes}(C) \cup \text{JoinNodes}(C).$$

Each preemption edge has a tuple of preemption type $k \in K := \{s, w, e\}$ and signal symbol $s \in S$, where $S$ is defined as the set of signal symbols. The preemption type indicates whether the preemption results from strong abort, weak abort or an exit, and the signal symbol by which signal the preemption is triggered.

$$P \subseteq V \times (K \times S) \times V.$$

Note that exit preemptions never occur for signals, but for trap labels, which are handled like signals by using signal symbols for them. Given a node $n \in V$, the set of its control flow successors, $n.\text{suc}_c \in V$, is the image of $E$ under $n$:

$$n.\text{suc}_c := E[n] = \{m \mid (n, m) \in E\}.$$

The preemption successors are defined the same way for all $k \in \{s, w, e\}$ by:

$$n.\text{suc}_k := P[n]_{(k,)} := \{m \in V \mid \exists s \in S : (n, m)_{(k,s)} \in P\}.$$

Furthermore the successors $n.\text{suc}$ and the instantaneous successors $n.\text{suc}_{\text{inst}}$ of $n \in V$ are defined by:

$$n.\text{suc} := E[n] \cup P[n]$$

$$n.\text{suc}_{\text{inst}}.$$
2. Instantaneous Paths

and

\[ n.suc_{\text{inst}} := \begin{cases} n.suc_c & : n \in \text{TransientNodes} \cup \text{LabelNodes} \\ n.suc_w \cup n.suc_e & : n \in \text{DelayNodes} \end{cases} \]

Note that transient nodes/label nodes have no preemption successors and its control flow successors are always instantaneous:

\[ n \in \text{TransientNodes} \cup \text{LabelNodes} : n.suc = n.suc_{\text{inst}} = E[n] \]

To the contrary the instantaneous successors of delay nodes are never control flow successors, but all preemption successors \((n.suc_w \cup n.suc_e \subseteq P[n])\). The preemption edges of types \(w\) and \(e\) are instantaneously executed.

2.2. Sequential Paths

A sequential path \(t\) in \(C\) with length \(|t| := l \geq 0\) is a sequence \(t = (t_1, \ldots, t_l)\) of nodes \(t_i \in V\) that \(t\) forms a chain in \(C\), i.e.:

\[ \forall i \in \mathbb{N}_{<l} : t_{i+1} \in t_i.suc. \]

The set of all sequential paths of \(C\) are denoted with \(P_{\text{seq}}(C)\).

\(t\) is called instantaneous iff

\[ \forall i \in \{1, \ldots, l\} : t_{i+1} \in t_i.suc_{\text{inst}}. \]

The set of all instantaneous sequential paths of \(C\) is denoted with \(I_{\text{seq}}(C)\).

Given \(t = (t_1, \ldots, t_l) \in I_{\text{seq}}(C)\), we define \(t\) as not extensible, if no \(t' \in I_{\text{seq}}(C)\) exists with \(t\) being a real sub-path of \(t'\):

\[ t \text{ not extensible} := \forall t' \in I_{\text{seq}}(C) : t \not< t' \]

whereby the sub-path relation is defined via

\[ t < t' := [ (\ldots, t_1, \ldots, t_l, \ldots) = t' \land |t| < |t'| \] \]

Given a not extensible path \(t\), all its statements form a possible KEP execution trace, because due to the KEP semantics all instantaneously reachable statements can be executed. We call \(t\) in this case a trace or reachable path. To determine not extensible instantaneous paths that are no traces is hard in general. Therefore we consider all paths as reachable.

A simple sequential Esterel example \(\text{ExSeq}\) can be found in Figure 2.1(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.
2.3. Parallel Paths

The CKAG of this example has five not extensible instantaneous paths, four of them are traces as shown in Figure 2.1(d):

\[(\text{WABORT}_{L1}, \text{A1}_{L2}, \text{PAUSE}_{L2}), (\text{PAUSE}_{L2}, \text{EMIT}_{L3}, \text{GOTO}_{L4}, \text{A1}_{L2}, \text{PAUSE}_{L2}), \]
\[(\text{PAUSE}_{L2}, \text{EMIT}_{L3}, \text{GOTO}_{L4}, \text{A1}_{L2}, \text{PAUSE}_{L2}, \text{A0}_{L5}, \text{EMIT}_{L5}, \text{HALT}_{L6}), (\text{HALT}_{L6}). \]

The remaining path \( r := (\text{WABORT}_{L1}, \text{A1}_{L2}, \text{PAUSE}_{L2}, \text{A0}_{L5}, \text{EMIT}_{L5}, \text{HALT}_{L6}) \) is not a trace, because of the Esterel semantics the weak abort edge \( \text{(PAUSE}_{L2}, \text{A0}_{L5}) \) cannot be executed in the same instant like the \( \text{WABORT}_{L1} \), since this is not an immediate abort. So path \( r \) is not reachable, nevertheless the WCRT analysis will conservatively consider such paths to get an algorithm of polynomial complexity.

2.3. Parallel Paths

To match concurrent control flow the concept of sequential paths is enhanced to the more general idea of parallel paths. We define the set of parallel paths \( P_{\text{par}}(C) \) of a CKAG \( C \) by using recursively the parallel operator \( \| \) to define two paths as concurrent to each other and the sequential operator \( ; \) to connect two paths sequentially. A parallel path is either a sequential path, the parallel of parallel paths or the sequence of parallel paths:

\[ P = S \quad | \quad (P \| P)_j \quad | \quad (P ; P) \]
2. Instantaneous Paths

whereby \( j \in \text{JoinNodes} \) denotes the join node the \( || \) belongs to.

Similar to the definition of instantaneous sequential paths in Section 2.2 we define instantaneous parallel paths as a parallel paths whose statements are possibly executed within the same instant:

\[
P_{\text{inst}} = S_{\text{inst}} \mid (P_{\text{inst}} || P_{\text{inst}})_j \mid (P'_{\text{inst}} ; P_{\text{inst}})
\]

whereby \( P'_{\text{inst}} \) is defined as a parallel paths whose concurrent paths merge to some control flow after the parallel specific JOIN. That is the case if for each concurrent flow all sub-thread paths terminate at their join node (see Path 2 in Figure 2.2(d)) or a thread exits by an instantaneous preemption path which results in the termination of the whole parallel statement (Path 1b). Note that \( P'_{\text{inst}} \) is introduced because in a path of kind \( (P_{\text{inst}} || P_{\text{inst}})_j \) (see Path 1a in Figure 2.2(d)) the execution of the parallel may need several ticks. So paths of sort \( ((P_{\text{inst}} || P_{\text{inst}})_j ; P_{\text{inst}}) \) are no instantaneous parallel paths in general.

The set of all instantaneous parallel paths is denoted by \( I_{\text{par}}(C) \). Not extensible parallel paths from \( I_{\text{par}}(C) \) are possible parallel traces.

2.4. Fork-Join Reachability

During the WCRT analysis all instantaneous paths have to be detected, in particular paths whose control flow continues after a concurrent execution has to be detected or excluded. This problem resembles the question whether a given fork-join node pair \((f, j)\) is part of a parallel path of kind \( P'_{\text{inst}} \) defined in Section 2.3. If such a path exists, we say \((f, j)\) is instantaneous, which means \((f, j)\) may be executed instantaneously. Instantaneous Reachability is now defined more exactly together with the additional non-instantaneous:

\[
\begin{align*}
(f, j) \text{ instantaneous} & : \exists t \in I_{\text{par}} : t = (\ldots \text{PAR*} ; (\ldots || \ldots) \text{JOIN} ; \ldots) \\
(f, j) \text{ non-instantaneous} & : \exists t \in I_{\text{par}} : t = (\ldots \text{PAR*} ; (\ldots || \ldots) \text{JOIN}) \lor \\
& \quad t = (\ldots || \ldots) \text{JOIN} [;\ldots]
\end{align*}
\]

whereby \( f.\text{stmt} = \text{PAR*} \) and \( j.\text{stmt} = \text{JOIN} \).

Note that in path \((\ldots \text{PAR*} ; (\ldots || \ldots)_\text{JOIN} ; \ldots)\) the parallel is started and again ended instantaneously.

If \((f, j)\) is instantaneous and not non-instantaneous than \((f, j)\) is always instantaneously executed. On the other hand if \((f, j)\) is not instantaneous and non-instantaneous than \((f, j)\) is always executed with a delay. If \((f, j)\) is both, then \((f, j)\) may be executed instantaneously but need not.

Next we consider reachability in general and the algorithm to computate it.

2.5. General Statement Reachability

We define two KEP statements instantly reachable if an instantaneous path exists that contains both statements (or rather their CKAG nodes) and delayed oth-
2.5. General Statement Reachability

erwise. The basic idea of the algorithm is to compute for each node three potential reachability properties: *instantaneous*, *non-instantaneous*, *exit-instantaneous*. Property *exit-instantaneous* is a special kind of *instantaneous* which occurs by instantaneous preemption, both of them are standing for *instantaneously reachable*. Note that *delayed* corresponds to the absence of both instantaneous properties, and not to *non-instantaneous*. 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 $O(3 \times |\text{Nodes}|) = O(|\text{Nodes}|)$.

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. While the parallel may be left immediately by an *EXIT* statement, the further behavior differs. Therefore we use the third property for this, beside *instantaneous* and *non-instantaneous*: *exit-instantaneous*.

Roughly speaking the *instantaneous* property is propagated via a for-all quantifier, *non-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.
2. Instantaneous Paths

Figure 2.2: An Esterel example with concurrency (a), the resulting KEP assembler program (b) and CKAG (c) to show different kinds of instantaneous parallel paths (d). These paths are not extensible and reachable: signal A has to be present and absent respectively to reach path 1a and 1b respectively. So these paths are traces.
3. Worst Case Reaction Time Analysis

Given a KEP program we define its WCRT as the maximum number of KEP cycles executable in one instant. Thus the WCRT analysis requires to find the longest instantaneous path in the CKAG, measured in the number of required KEP instruction cycles. Next will be described how these cycles are computed. Hereafter we present a restricted form of the WCRT algorithm that handles only sequential programs, which is then generalized. The general algorithm handles additionally fork and join nodes and requires the analysis of instantaneous reachability between fork and join nodes, which was discussed in Section 2.4. The general WCRT algorithm is presented in Section 3.3. The algorithms abstract from signal relationships and might therefore consider unfeasible executions. Therefore the computed WCRT is in general pessimistic. Such WCRT overestimations are discussed in Section 3.4.

3.1. KEP Instruction Cycles

Next we describe how to compute the KEP instruction cycles we need to measure the length of a path of nodes. The cycles of a node are simply defined as the cycles of the statements they contain:

\[ n \in \text{Nodes} : cycles(n) := cycles(n.stmt). \]

Most nodes contain exactly one statement, which needs exactly one instruction cycle, except for the following cases:

- The root node contains statements which define the programs interface and variables with INPUT/OUTPUT and VAR respectively. As an optimization local SIGNAL statements are added when they occur globally. All these statements are used to initialize the program before the real program starts, so all cycles of statements in the interface are defined as zero.

- The amount of KEP statement cycles of a fork node \( f \) depends on the number of its sub-threads. For each sub-thread a PAR is executed for its initialization (see Figure 1.1). With the PARE execution at the end of thread initialization we obtain

\[ cycles(f.stmt) = cycles(\text{PAR}^*) = \sum_{i \in \text{f.threads}} cycles(\text{PAR}_i) + cycles(\text{PARE}). \]
3. Worst Case Reaction Time Analysis

- In the case of nested concurrency all involved join statements are executed a second time, according to their hierarchy from innermost to outermost. This ensures a correct termination order when exits occur. So the cyclecount of a JOIN varies from one to two instruction cycles.

- A label node contains no real KEP statement, it contains an address label which needs zero instruction cycles.

- A function call needs one instruction cycle, but the WCRT of the function that is called depends on the function, so it varies and has to be computed separately. Our WCRT implementation requires that the result is be printed within a specific KEP function file in the format "%%% WCRT: <wcrt>" as commentary, to be parsed and used during the WCRT analysis.

The amount of KEP instruction cycles of a path \( t \in I_{seq}(C) \) is defined as the sum of the individual statement cycles:

\[
cycles(t) := \sum_i \cycles(t_i)
\]

and for \( t \in I_{par}(C) \) we define:

\[
cycles((P_1 || \ldots || P_n)_j) = \sum_i \cycles(P_i) + \cycles(j)
\]

\[
cycles(P_1 ; P_2) = \cycles(P_1) + \cycles(P_2)
\]

whereby \( t = (P_1 || \ldots || P_n)_j \) or \( t = (P_1 ; P_2) \) respectively.

Note that the amount of instruction cycles of a statement depends on KEP semantics and may vary when the KEP implementation changes. But this will not effect the algorithm, because this is scalable by hiding the computation of statement cycles in the function \( \cycles \).

3.2. Sequential WCRT Analysis

First we present a WCRT analysis of sequential CKAGs (no fork and join nodes).

Consider again the ExSeq example in Figure 3.1(a). The longest possible execution occurs when the signal \( I \) becomes present, as is the case in Tick 3 of the example trace shown in Figure 3.1(d). Since the abortion triggered by signal \( I \) is weak, the abort body is still executed in this instant, which takes four instructions: PAUSE_L2, EMIT_L3, the GOTO_L4, and PAUSE_L2 again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and EMIT_L5 and HALT_L6 are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via an instantaneous Depth First Search (DFS) traversal of the CKAG, see the algorithm in Figure 3.2. For each node \( n \) a value \( n.inst \) is computed, which gives the WCRT from this node on in the same instant.
3.3. General WCRT Analysis

when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For each non-instantaneous delay node \( d \) an additional value \( d.next \) stores the maximal number of instantaneously reachable instructions, when the execution starts at \( d \). These two values are needed to ensure 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 or continuations 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 instant the delay node is reached. So the WCRT of the handler of a strong abortion is added to \( d.next \). We do not need to take a weak abortion into account here, because it cannot contribute to a longest path. An abortion in the first instant will always lead to a higher WCRT than an execution that starts at 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 again the example \( \text{ExSeq} \) in Figure 3.1. Each node \( n \) in the CKAG \( g \) is annotated with a label \( "W(n.inst)" \) or, for a delay node, a label \( "W(n.inst)/(n.next)".\) In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers \( L(n) \). It is \( g.root = L1 \). The sequential WCRT computation starts initializing the \( inst \) and \( next \) values of all nodes to \( \bot \) (line 2 in getWcrtSeq, Figure 3.2). Then \( \text{getInstSeq}(L1) \) is called, which computes \( L1.inst := \max \{ \text{getInstSeq}(L2) \} + \text{cycles}(\text{WABORT}_{L2}) \). The call to \( \text{getInstSeq}(L2) \) computes and returns \( L2.inst := \text{cycles}(\text{PAUSE}_{L2}) + \text{cycles}(\text{EMIT}_{L5}) + \text{cycles}(\text{HALT}_{L6}) = 3 \), hence \( L1.inst := 3 + 2 = 5 \). Next, in line 4 of getWcrtSeq, we call \( \text{getNextSeq}(L2) \), which computes \( L2.next := \text{getInstSeq}(L3) + \text{cycles}(\text{PAUSE}_{L2}) \). The call to \( \text{getInstSeq}(L3) \) computes and returns \( L3.inst := \text{cycles}(\text{EMIT}_{L3}) + \text{cycles}(\text{GOTO}_{L4}) + L2.inst = 1 + 1 + 3 = 5 \). Hence \( L2.next := 5 + 1 = 6 \), which corresponds to the longest path triggered by the presence of signal 1, as we have seen earlier. The WCRT analysis therefore inserts an “\text{EMIT} _TICKLEN, #6” instruction before the body of the KEP assembler program to initialize the \text{TickManager} accordingly [17].

3.3. General WCRT Analysis

The general algorithm shown in Figure 3.4 can also handle concurrency: it emerges from the sequential algorithm that has been described in Section 3.2 by enhancing it with the ability of computing the WCRT of fork and join nodes. Note that the instantaneous WCRT of a join node is started only by a fork node, all transient nodes and delay nodes do not use this value for their WCRT. This allows the use of a DFS like algorithm and the WCRT of the join node will be accounted just once in the instantaneous WCRT of its corresponding fork node.
3. Worst Case Reaction Time Analysis

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.

The join nodes, like delay nodes, also have a next value. When a fork-join pair \((f, j)\) could be non-instantaneous we have to compute a WCRT \(j.\text{next}\) for the next instants analogously to the delay nodes. Its computation requires first the computation of all sub-thread next WCRTs. Note that in case of nested concurrency these next values can again result from a join node. But at the innermost level of concurrency the next WCRT values all occur from delay nodes, which will be computed before the join next values. The delay next WCRT values are computed the same way as in the sequential case except that only successors within of the same thread are mentioned. We call successors of a different thread inter-thread-successors and their WCRT values are handled by the according join node. The join next value is the maximum of all inter-thread-successor WCRT values and the sum of the maximum next value for every thread. See the abro example in Figure 3.5 two inter-thread-successors are present because a parallel construct exists within the abort body.

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.

Figure 3.1.: The sequential Esterel example shown in Figure 2.1 (a), the corresponding CKAG annotated with the results of its WCRT analysis (b), the generated KEP completed with the initialization of the TickManager (c), and a sample execution trace (d). The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles.
3.3. General WCRT Analysis

```
int getWcrtSeq(g) // Compute WCRT for sequential CKAG g
forall n ∈ Nodes do n.inst := n.next := ⊥ end
getInstSeq(g.root)
forall d ∈ DelayNodes do getNextSeq(d) end
wcrt := max ({g.root.inst} ∪ {d.next : d ∈ DelayNodes})
return wcrt
end

int getInstSeq(n) // Compute statements instantaneously reachable from node n
if n.inst = ⊥ then
  if n ∈ TransientNodes ∪ LabelNodes then
    n.inst := max {getInstSeq(c) : c ∈ n.suc} + cycles(n.stmt)
  elsif n ∈ DelayNodes then
    n.inst := max {getInstSeq(c) : c ∈ n.sucw ∪ n.suce} + 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 = ⊥ then
  d.next := max {getInstSeq(c) : c ∈ d.sucw ∪ d.suce} + cycles(d.stmt)
fi
return d.next
end
```

Figure 3.2.: WCRT algorithm, restricted to sequential programs. Function return values may be ignored. The nodes of a CKAG g are given by Nodes = TransientNodes ∪ LabelNodes ∪ DelayNodes ∪ ForkNodes ∪ JoinNodes, g.root indicates the first KEP statement. cycles(stmt) returns the number of instruction cycles to execute stmt, see third column in Figure 1.1.

The complete 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 3.3. First we note that the fork/join pair is always non-instantaneous. We compute L6.next = cycles(PAUSEL6) + cycles(EIMITL7) = 2.

From the fork node L3, the PAR and PARE statements, the instantaneous parts of both threads and the JOIN are executed, hence L3.inst = 2 × cycles(PAR) + cycles(PARE) + cycles(JOIN) + L4.inst + L5.inst = 7. Therefore, the WCRT of the program is L8.next = L6.next + L8.inst = 2 + 9 = 11. Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel-programs is that due to the possible 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 \( n := |\text{Nodes}| \), \( d := |\text{DelayNodes}| \), \( f := |\text{ForkNodes}| \), and \( j := |\text{JoinNodes}| \). For each node its WCRT’s \( \text{inst} \) and \( \text{next} \) are computed at most once, and for all fork nodes a fork-join reachability analysis is additionally performed, which has itself \( O(n) \). So we get altogether a complexity of \( O(n + d + j) + O(f \ast n) = O(2 \ast n) + O(n^2) = O(n^2) \).

### 3.4. WCRT Overestimation

In the algorithms described before, signal informations are not taken into account. This can lead to an overestimation of the WCRT, because unreachable paths are considered in the analysis which can never be executed. Paths that contain dead code are trivially unreachable and this problem could be solved by a dead code analysis \[27\]. In Figure 3.6 we see an unreachable path increasing needlessly the WCRT because it assumes signal \( I \) present and absent instantaneously, which is inconsistent. Nevertheless there is no dead code in the CKAG but only two possible paths regarding the path signal predicates.
3.5. Experimental Results

Figure 3.7 shows an unreachable parallel path that leads to a too high WCRT of the fork node:

- the sub-paths cannot be executed at the same time, and
- the parallel is declared as possibly instantaneous, even though it is not. Therefore, all statements which are instantaneously reachable from the join node are also added.

Another unreachable parallel path is shown in Figure 3.8. This path is unreachable not because of signal informations but because of instantaneous behavior: the maximal paths of the two threads are never executed in the same instant. Instead of summing up for each thread the maximum next WCRT it would be more exact to sum up over all threads next WCRT’s that are executable instantaneously and then taking the maximum of these sums. To perform this analysis we would have to enhance the reachability algorithm with the ability to determine how many ticks later a statement could be executed after each other. The possible tick counts can reach arbitrary values for each node, so we would get a higher complexity and a determination problem. Our analysis is conservative in simply assuming that all concurrent paths may occur in the same instant, and that all can be executed in the same instant as the join.

3.5. 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 3.1 are taken from the Estbench [6]. We never underestimated the WCRT, and our results are on average 24% 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 execution time on a standard PC (AMD Athlon XP, 2.2GHz, 512 KB Cache, 1GB Main Memory) and the number of scenarios and accumulated logical tick count for the test traces. As the table indicates, the analysis takes only a couple of milliseconds.
### 3. Worst Case Reaction Time Analysis

<table>
<thead>
<tr>
<th>module name</th>
<th>LoC</th>
<th>WCRT Estimated</th>
<th>WCRT Measured</th>
<th>WCRT Overest. [%]</th>
<th>ACRT utilization [%]</th>
<th>Test cases</th>
<th>Ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd</td>
<td>152</td>
<td>47</td>
<td>44</td>
<td>7%</td>
<td>27</td>
<td>61%</td>
<td>161</td>
</tr>
<tr>
<td>abcdef</td>
<td>232</td>
<td>71</td>
<td>68</td>
<td>4%</td>
<td>41</td>
<td>60%</td>
<td>1457</td>
</tr>
<tr>
<td>eight_buttons</td>
<td>332</td>
<td>96</td>
<td>92</td>
<td>4%</td>
<td>57</td>
<td>62%</td>
<td>13121</td>
</tr>
<tr>
<td>channel_protocol</td>
<td>57</td>
<td>41</td>
<td>38</td>
<td>8%</td>
<td>18</td>
<td>47%</td>
<td>114</td>
</tr>
<tr>
<td>reactor_control</td>
<td>24</td>
<td>17</td>
<td>14</td>
<td>21%</td>
<td>10</td>
<td>71%</td>
<td>6</td>
</tr>
<tr>
<td>runner</td>
<td>26</td>
<td>12</td>
<td>10</td>
<td>20%</td>
<td>2</td>
<td>20%</td>
<td>131</td>
</tr>
<tr>
<td>ww_button</td>
<td>94</td>
<td>31</td>
<td>18</td>
<td>72%</td>
<td>12</td>
<td>67%</td>
<td>8</td>
</tr>
<tr>
<td>tcint</td>
<td>410</td>
<td>192</td>
<td>138</td>
<td>39%</td>
<td>86</td>
<td>62%</td>
<td>148</td>
</tr>
</tbody>
</table>

Table 3.1.: Comparing the WCRT estimated by our analysis with the actual WCRT. ACRT is the Average Case Reaction Time. Test cases and Ticks are the number of different scenarios and logical ticks that were executed, respectively.
3.5. Experimental Results

```c
int getWcrt(g) // Compute WCRT for a CKAG
g
for all n ∈ Nodes do n.inst := n.next := ⊥ end
for all d ∈ DelayNodes do getNext(d) end
for all j ∈ JoinNodes do getNext(j) end // Visit according to hierarchy (inside out)
wcrt := max \{ (getInst(g.root)) ∪ \{ n.next : n ∈ DelayNodes ∪ JoinNodes)) \}
return wcrt
end

int getInst(n) // Compute statements instantaneously reachable from node n
if n.inst = ⊥ then
if n ∈ TransientNodes ∪ LabelNodes then
t.inst := max \{ getInst(c) : c ∈ suc_c \ JoinNodes \} + cycles(nstmt)
elif n ∈ DelayNodes then
n.inst := max \{ getInst(c) : c ∈ suc_c ∪ suc_e \ JoinNodes \} + cycles(nstmt)
elif n ∈ ForkNodes then
n.inst := \sum_{t ∈ n.suc} 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 ∈ JoinNodes then
n.inst := max\{ getInst(c) : c ∈ suc_e ∪ suc_e \} + cycles(nstmt);
fi
fi
return n.inst
end

int getNext(n) // Compute statements instantaneously reachable from delay node d at tick start
if n.next = ⊥ then
if n ∈ DelayNodes then
n.next := max \{ getInst(c) : c ∈ suc_e ∪ suc_e \ JoinNodes ∧ c.id = n.id \} + cycles(nstmt)
// handle inter thread successors by their according join nodes:
for m ∈ \{ c ∈ suc_e ∪ suc_e \ JoinNodes : c.id = n.id \} do
j := according join node with j.id = m.id
j.next := max \{ j.next, getInst(m) + cycles(mstmt) + cycles(jstmt) \}
end
elif n ∈ JoinNodes then
prop := reachability(n.fork, n) // Compute reachability predicates
if prop.not_instantaneous then
n.next := max \{ \sum_{t ∈ n.fork.suc} max\{ m.next : t.id = m.id \} + n.inst \ n.next \}
fi
fi
fi
return n.next
end

Figure 3.4.: General WCRT algorithm.
3. Worst Case Reaction Time Analysis

![Diagram](image.png)

**Figure 3.5.** The standard ABRO example: there are strong abort edges between the delay nodes with AWAIT statements to a node outside the parallel indicated by a different thread-id. So their next WCRT values are used for next value of the join node.

![Diagram](image.png)

**Figure 3.6.** Inconsistent sequential path example: signal I cannot be present and absent within the same tick due to Esterel semantics, so this program has two inconsistent paths.
3.5. Experimental Results

Figure 3.7: Inconsistent parallel path example

Figure 3.8: Unreachable parallel path example: a more exact reachability analysis would be necessary to determine whether statements of different threads are executable within the same instant.
3. Worst Case Reaction Time Analysis
4. Implementation

The WCRT analysis is implemented within the Esterel KEP compiler which is based on the CEC [3]. The implementation consists of two C++ files: WCRT.hpp and WCRT.cpp. The header file WCRT.hpp contains the definition of classes and their data fields and method prototype declarations. In file WCRT.cpp the bodies of the method prototypes defined in WCRT.hpp are implemented. Next the defined classes are described in detail and how they are used to perform the WCRT analysis.

Three classes are used to implement the WCRT analysis: WCRT, WCRT_Sequential and WCRT_Parallel. WCRT_Parallel is a subclass of WCRT_Sequential, which is in turn a subclass of WCRT.

4.1. class WCRT

Class WCRT is a subclass of class KEP::Visitor implemented in header file KEP.hpp which provides for all existing KEP data structures a default visit method building up the framework for all visitor [12] implementations.

The computation of KEP statement cycles is realized by overriding the default KEP::Visitor visit methods for all KEP statements. Each of these visit methods sets the class variable cycles to the statement specific value. If a statement is not supported the visit method of its parent class is called, which throws an assertion.

Class WCRT also implements the basic data structures and methods needed by the algorithm, especially the data fields needed to store the inst and next WCRT values for each node:

```cpp
typedef std::map<KEP::KepNode*,unsigned int> wcrt_map;
typedef struct ReactionTimes {
    wcrt_map inst;
    wcrt_map next;
};;
ReactionTimes wcrt;
```

To ensure the termination of the algorithm it is essential to avoid the cyclic visiting of nodes. Therefore the nodes which the algorithm is currently visiting is saved in a data structure called NodeVisit:

```cpp
typedef struct NodeVisit {
    std::set<KEP::KepNode*> inst;
    std::set<KEP::KepNode*> next;
};
```
4. Implementation

NodeVisit visiting;

whereby two visits are allowed if in one case the inst in the other the next value is needed. Structure NodeVisit takes this into account by having for both cases an own node set. If a node is visited to compute a wcrt value \( v \in \{ \text{inst}, \text{next} \} \) which is already visited for the same value \( v \), the boolean variable error is set to true which stops a further WCRT computation. When an error occurs the WCRT is set to the bottom value zero to indicate that it was not possible to compute the WCRT for the given program/CKAG.

4.2. class WCRT_Sequental

Class WCRT_Sequental implements the sequential WCRT algorithm via a visitor pattern [12] through visit methods for transient nodes, label nodes and delay nodes. The algorithm described in Figure 3.2 contains if-elif-else conditional constructs to distinguish different kinds of nodes, which are each represented by a visit method for each kind, which contains of the specific conditional body. A node welcomes the visitor and then the according visit method is executed. This mechanism is explained in more detailed in Figure 4.1. If a fork or join node occurs, a warning is thrown, because such kind of nodes are only supported by the general WCRT algorithm implemented by class WCRT_Parallel.

To determine within the visit methods the kind of WCRT value that has to be computed, inst or next, this class provides a data field mode of type enum:

typedef enum { INST,NEXT,UNDEFINED } wcrt_mode;

wherby the occurrence of UNDEFINED would indicate a bug in the implementation and kind NEXT is ignored by visit methods of nodes that have no next WCRT value to be computed like transient nodes. Therefore the computation of all next values is easily implemented by visiting all nodes in mode NEXT.

4.3. class WCRT_Parallel

Class WCRT_Parallel inherits the sequential algorithm described in Figure 3.2 from class WCRT_Sequental, but is designed to implement the general algorithm of Figure 3.4. To accomplish this, some of the derived methods are overridden and/or enhanced. For example to handle preemption edges between different threads starting from delay nodes, the visit method for delay nodes has to be overridden, because this kind of edge does not exist in a sequential CKAG and is therefore not mentioned in the sequential version. For the same reason the data field join_outer_next is added to store for each join node the maximum WCRT value, which originates from these inter-thread edges.
Class \texttt{WCRT\_Parallel} additionally implements visit methods for the \textit{fork} and \textit{join node} by overriding the dummy methods of \texttt{WCRT\_Sequential}. Thus this class handles concurrency and all kind of nodes are supported. To compute the next value of a \textit{join node} all maximum next values within its sub-threads are needed, so we save them in data field \texttt{max\_inner\_next} of type \texttt{id\_map}:

\begin{verbatim}
typedef std::map<KEP::KepThreadId*,unsigned int> id_map;

id_map max\_inner\_next;
\end{verbatim}


to sum them up. Note that only next values from sub-threads of depth one are added: sub-sub-thread values are handled by their according join nodes, whose thread identifiers are just one level higher. Hereby the deeper values are considered by taking the join next values also into account.

Data structure \texttt{max\_inner\_next} is initialized with zero values to ensure a defined value and avoid testing for it. If there are no next values within a thread, the adding of zero has no impact to the resulting maximum inner next sum.

4.4. Usage

The WCRT implementation within other C++ modules in particular is used within the \texttt{strl2kep} compiler to compute the WCRT of a given Esterel program. During the analysis all WCRT values are saved and are displayed by using specific options. This is useful for debugging and documentation purposes.
4. Implementation

4.4.1. strl2kep compiler

The WCRT analysis is realized in file `cec-astkep.cpp` during the compilation from an Esterel AST to KEP. An instance of class `WCRT_Sequential` or `WCRT_Parallel` is created according to which kind of algorithm has to be called, the sequential or the general one. Then the method `compute` with the current CKAG as argument is called:

```
// WORST CASE REACTION TIME (WCRT) ANALYSIS
d->print("start worst case reaction time analysis:");
time_measurement.start();

// wcrt analysis
WCRT::WCRT_Parallel wcrt(d);
wcrp.compute(ckag);

// save wcrt time informations
time_measurement.end();
time_measurement.addDuration("WCRT","WCRT TIME:");
```

whereby for documentation purposes the analysis time is measured by saving the time at start and computing the difference at the end. In Table 3.1 the results for several Esterel examples are shown.

4.4.2. Compiler Options

The `strl2kep` compiler has implemented two pretty printers: `cec-xmlkasm` to print the resulting KEP assembler program and `cec-kepdot` to print the corresponding CKAG graph in the `dot` format. Both printers have several printing options to control the appearance of the result and to add informations for documentation or debugging reasons.

To add the computed WCRT values to the statements respectively nodes, the `print` option `-p <kind>` can be used with print kinds "WCRT" or "ALL":

- `cec-xmlkasm -p "WCRT"`: The WCRT values of the nodes are added to their statements within a comment block of square brackets. Statements with no according CKAG node are left blank instead of a value. Typically the PAR statements have no value, because the WCRT value of the according fork node is added to the PARE statement as its representative.

- `cec-kepdot -p "WCRT"`: For each node its WCRT values are added to their dot labels. Additionally the result is printed to the root node by adding the EMIT statement which sets variable `_TICKLEN`.

The WCRT values are printed in the following format: `W<inst>[/<next>]` whereby for join nodes the occurrence of the next depends on their reachability status as mentioned in Section 2.4. The WCRT printing is used in most of the examples that make use of a KEP program or CKAG graph.
5. Conclusions and Further Work

I 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 terminate instantaneously, thereafter we are able to compute for each statement how many instructions are maximally executable from it in one logical tick. The maximal value of 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.

The 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 25%. 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 grow 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.
5. Conclusions and Further Work
6. Bibliography


6. Bibliography


6. Bibliography


6. Bibliography
7. Appendix

A. C++ Sources
A.1. WCRT.hpp

```cpp
#include <iostream>
#include <fstream>
#include <cassert>
#include <queue>
#include "IR.hpp"
#include "AST.hpp"
#include "KEP.hpp"
#include "CKAG.hpp"
#include "KepHandler.hpp"

namespace WCRT {
  /******************************
  *** W C R T
  ******************************/

  // class WCRT::WCRT
  class WCRT : public KEP::Visitor {
    protected:
      KEP::CKAG *ckag;
      KEP::Debug *debug;
      typedef std::map<KEP::KepNode*,unsigned int> wcrt_map;
      typedef struct ReactionTimes {
        wcrt_map inst;
        wcrt_map next;
      } ReactionTimes;
      ReactionTimes wcrt;
      typedef struct NodeVisit {
        std::set<KEP::KepNode*> inst;
        std::set<KEP::KepNode*> next;
      } NodeVisit;
      NodeVisit visiting;
      bool error;
      unsigned int cycles;
    public:
      WCRT(KEP::Debug *d = new KEP::Debug()) : ckag(NULL), debug(d), wcrt(), visiting(), error(false), cycles(0) {}
      virtual ~WCRT() {};
      virtual bool compute(KEP::CKAG*);
      void collect(void);

      protected:
        // NODES
        virtual void visit(KEP::TransientNode&);
        virtual void visit(KEP::LabelNode&);
        virtual void visit(KEP::DelayNode&);
        virtual void visit(KEP::ForkNode&);
        virtual void visit(KEP::JoinNode&);
        virtual void visit(KEP::Interface&);
        virtual void visit(KEP::Address&);
        virtual void visit(KEP::Var&);
        virtual void visit(KEP::Nothing&);
        virtual void visit(KEP::Clrc&);
        virtual void visit(KEP::Setc&);
        virtual void visit(KEP::Goto&);
        virtual void visit(KEP::Exit&);
        virtual void visit(KEP::Emit&);
        virtual void visit(KEP::Present&);
        virtual void visit(KEP::Signal&);
        virtual void visit(KEP::Setv&);
        virtual void visit(KEP::Jw&);
        virtual void visit(KEP::RegisterOp&);
        virtual void visit(KEP::RegisterDataOp&);
        virtual void visit(KEP::Def32&);
        virtual void visit(KEP::Abort&);
        virtual void visit(KEP::Suspend&);
        virtual void visit(KEP::Delay&);
        virtual void visit(KEP::Halt&);
        virtual void visit(KEP::Await&);
        virtual void visit(KEP::Sustain&);
        virtual void visit(KEP::Par&);
        virtual void visit(KEP::Pare&);
        virtual void visit(KEP::Prio&);
        virtual void visit(KEP::Join&);
        virtual void visit(KEP::KepFunctionCall&);
        virtual void visit(KEP::KepReturn&);

      // STATEMENTS
      virtual void visit(KEP::Instruction&);
      virtual void visit(KEP::Assign&);
      virtual void visit(KEP::LogicalOp&);
      virtual void visit(KEP::ArithmeticOp&);
      virtual void visit(KEP::BitwiseOp&);
      virtual void visit(KEP::StringOp&);
      virtual void visit(KEP::BooleanOp&);
      virtual void visit(KEP::Cast&);
      virtual void visit(KEP::FunctionCall&);
      virtual void visit(KEP::FunctionReturn&);

      // DELAY
      virtual void visit(KEP::Delay&);
      virtual void visit(KEP::Halt&);
      virtual void visit(KEP::Await&);
      virtual void visit(KEP::Sustain&);
      virtual void visit(KEP::Par&);
      virtual void visit(KEP::Pare&);
      virtual void visit(KEP::Prio&);
      virtual void visit(KEP::Join&);
      virtual void visit(KEP::KepFunctionCall&);
      virtual void visit(KEP::KepReturn&);

      protected:
        virtual void visit(KEP::Interface&);
        virtual void visit(KEP::Address&);
        virtual void visit(KEP::Var&);
        virtual void visit(KEP::Nothing&);
        virtual void visit(KEP::Clrc&);
        virtual void visit(KEP::Setc&);
        virtual void visit(KEP::Goto&);
        virtual void visit(KEP::Exit&);
        virtual void visit(KEP::Emit&);
        virtual void visit(KEP::Present&);
        virtual void visit(KEP::Signal&);
        virtual void visit(KEP::Setv&);
        virtual void visit(KEP::Jw&);
        virtual void visit(KEP::RegisterOp&);
        virtual void visit(KEP::RegisterDataOp&);
        virtual void visit(KEP::Def32&);
        virtual void visit(KEP::Abort&);
        virtual void visit(KEP::Suspend&);
        virtual void visit(KEP::Delay&);
        virtual void visit(KEP::Halt&);
        virtual void visit(KEP::Await&);
        virtual void visit(KEP::Sustain&);
        virtual void visit(KEP::Par&);
        virtual void visit(KEP::Pare&);
        virtual void visit(KEP::Prio&);
        virtual void visit(KEP::Join&);
        virtual void visit(KEP::KepFunctionCall&);
        virtual void visit(KEP::KepReturn&);
      
    }; // class WCRT::WCRT
  }; // class WCRT
}; // namespace WCRT
```
/* auxiliary functions */

private:
bool has_inner_concurrency(KEP::ForkNode*);

/******************
*** SEQUENTIAL WCRT
******************/

class WCRT_Sequential : public WCRT {
protected:
typedef enum { INST, NEXT, UNDEFINED } wcrt_mode;
wcrt_mode mode;

public:
WCRT_Sequential(KEP::Debug *d = new KEP::Debug()) : WCRT(d), mode(UNDEFINED) {}

virtual ~WCRT_Sequential() {};

virtual bool compute(KEP::CKAG*);

protected:
virtual void visit(KEP::TransientNode&);
virtual void visit(KEP::LabelNode&);
virtual void visit(KEP::DelayNode&);
virtual void visit(KEP::ForkNode&);
virtual void visit(KEP::JoinNode&);

// AUXILIARY FUNCTIONS
protected:
virtual unsigned int compute_inst_wcrt(KEP::KepNode*);
virtual unsigned int compute_next_wcrt(KEP::KepNode*);
unsigned int max_wcrt(std::vector<KEP::KepNode*>);

};

class WCRT_Parallel : public WCRT_Sequential {
protected:
typedef std::map<KEP::KepThreadId*, unsigned int> id_map;

// save the maximum next value between different threads for each join node
std::map<KEP::JoinNode*, unsigned int> join_outer_next;

// save the maximum next value of a thread in id_map, needed during join next computation
id_map max_inner_next;

public:
WCRT_Parallel(KEP::Debug *d = new KEP::Debug()) : WCRT_Sequential(d) {};

virtual ~WCRT_Parallel() {};

virtual bool compute(KEP::CKAG*);

protected:
virtual void visit(KEP::TransientNode&);
virtual void visit(KEP::LabelNode&);
virtual void visit(KEP::DelayNode&);
virtual void visit(KEP::ForkNode&);
virtual void visit(KEP::JoinNode&);
virtual void visit(KEP::KepThreadId&);

// AUXILIARY FUNCTIONS
protected:
virtual unsigned int compute_inst_wcrt(KEP::KepNode*);
virtual unsigned int compute_next_wcrt(KEP::KepNode*);
unsigned int inner_thread_next_wcrt(KEP::KepNode*, std::vector<KEP::KepNode*>);

};

} // end of namespace WCRT in WCRT.hpp

#endif
#include "WCRT.hpp"

namespace WCRT {

/******************************/
*** WCRT
******************************/

bool WCRT::compute(KEP::CKAG *c) {
    assert(c);
    this->ckag = c;
    return (false);
}

void WCRT::visit(KEP::KepFunctionCall &c) {
    debug->print("start WCRT::visit(KepFunctionCall):","",c);
    debug->indent();
    unsigned int call_cycles = 1;
    KEP::FunctionSymbol *function = c.getFunctionSymbol();
    assert(function);
    std::string function_name = function->getName();
    debug->print("function:",function_name);
    unsigned int function_wcrt = 0;
    int kind = function->getKind();
    // INTERFACE FUNCTION (need external file)
    if (kind == KEP::FunctionSymbol::FUNCTION) {
        std::string file_name = c.getFileName();
        debug->print("try to open function file:",file_name);
        std::ifstream function_file;
        function_file.open(file_name.c_str());
        if (function_file.is_open()) {
            debug->print("file is open");
            unsigned int f_wcrt = 0;
            while (!function_file.eof() && function_file.good() && !f_wcrt) {
                std::string line;
                std::getline(function_file,line);
                // default, if parsing for wcrt not possible, we set variable parse to false
                bool parse = true;
                // search within comment
                const std::string str_comment = "%
                unsigned int pos = line.find_first_of(str_comment);
                if (parse && pos != std::string::npos) {
                    line = line.substr(pos + str_comment.size());
                } else {
                    parse = false;
                }
                // search for wcrt behind string "WCRT:
                const std::string str_search = "WCRT:
                pos = line.find_first_of(str_search);
                if (parse && pos != std::string::npos) {
                    line = line.substr(pos + str_search.size());
                } else {
                    parse = false;
                }
                if (parse) {
                    if (function_name == "DIV") {
                        function_wcrt += 443;
                    } else if (function_name == "MOD") {
                        debug->error("MOD WCRT NOT CHECKED YET!!!",c);
                        function_wcrt += 443;
                    } else {
                        debug->error("UNKNOWN BUILTIN FUNCTION",function);
                    }
                }
            }
            function_wcrt = this->parse_wcrt(function_file);
            function_file.close();
            debug->print("end WCRT::visit(FunctionCall),");
            this->cycles = call_cycles + function_wcrt;
            debug->unindent();
            debug->print("end WCRT::visit(FunctionCall)."c);
            this->cycles = call_cycles + function_wcrt;
            debug->print("end WCRT::visit(FunctionCall),");
            if (!function_wcrt) {
                debug->error("NO FUNCTION WCRT FOR",function);
                this->error = true;
            }
        } else {
            debug->error("KASM FILE " + file_name + " NOT FOUND TO PARSE THE WCRT FOR",c,
        true);
    }
    if (function_name == "DIV") {
        function_wcrt += 443;
    } else if (function_name == "MOD") {
        debug->error("MOD WCRT NOT CHECKED YET!!!",c);
        function_wcrt += 443;
    } else {
        debug->error("UNKNOWN BUILTIN FUNCTION",function);
    }
    this->cycles = call_cycles + function_wcrt;
    debug->print("end WCRT::visit(FunctionCall),");
    unsigned int WCRT::parse_wcrt(std::ifstream &f) {
        assert(f.is_open());
        unsigned int f_wcrt = 0;
        while (!f.eof() && f.good() && !f_wcrt) {
            std::string line;
            std::getline(f,line);
            if (f.is_open()) {
                debug->print("file is open");
                if (function_wcrt == 0) {
                    debug->warning("no WCRT found in file " + file_name);
                }
                function_file.close();
                debug->print("file is closed");
                if (function_name == "DIV") {
                    function_wcrt += 443;
                } else if (function_name == "MOD") {
                    debug->error("MOD WCRT NOT CHECKED YET!!!",c);
                    function_wcrt += 443;
                } else {
                    debug->error("UNKNOWN BUILTIN FUNCTION",function);
                }
            }
        }
    }
}

}
A. C++ Sources

```cpp
} else {
    parse = false;
    debug->warning("NO WCRT ASSIGNED BEHIND STRING "WCRT:");
}

// parse wcrt integer value
if (parse) {
    int done = sscanf(line.c_str(), "%u", &f_wcrt);
    if (done) {
        debug->print("parsing successful, function wcrt:", f_wcrt);
    } else {
        debug->error("WCRT PARSING ERROR");
    }
} else {
    // do not parse when string is empty
    if (parse && line.empty()) {
        parse = false;
        debug->warning("NO WCRT ASSIGNED BEHIND STRING "WCRT:");
    }
}

return (f_wcrt);
}

unsigned int WCRT::instruction_cycles(KEP::KepNode *n) {
    assert(n);
    this->cycles = 0;
    KEP::KepStatement *stmt = n->getStatement();
    assert(stmt);
    stmt->welcome(*this);
    unsigned int res = this->cycles;
    KEP::KepThreadId *id = n->getThreadId();
    assert(id);

    if (KEP::JoinNode *j = dynamic_cast<KEP::JoinNode*>(n)) {
        KEP::ForkNode *f_j = j->getForkNode();
        assert(f_j);
        if (id->getId()!=0) {
            debug->print("add one cycle to inner join node", j);
            res++;
        } else if (this->has_inner_concurrency(f_j)) {
            debug->print("add one cycle to main join node with inner join nodes", j);
            res++;
        } else {
            // do nothing: main join node without inner join nodes
        }
    }

    if (KEP::ForkNode *f = dynamic_cast<KEP::ForkNode*>(n)) {
        std::map<KEP::Par*, KEP::KepNode*> threads = f->getSubThreads();
        std::map<KEP::Par*, KEP::KepNode*>::iterator t;
        for (t=threads.begin(); t!=threads.end(); t++) {
            assert(t->first);
            if (t->first->getThreadId()) {
                debug->print("add one cycle to sub thread", t->first);
            }
        }
        res += this->cycles;
    }

    return (res);
}

bool WCRT::has_inner_concurrency(KEP::ForkNode *f) {
    assert(f);
    KEP::KepThreadId *id = f->getThreadId();
    assert(id);
    bool inner = true;
    assert(id->getId() >= 0);
    const unsigned int fork_id = (unsigned int)id->getId();
    std::vector<KEP::KepThreadId*> subthreads = f->getSubThreadIds();
    std::vector<KEP::KepThreadId*>::iterator i;
    for (i=subthreads.begin(); i!=subthreads.end(); i++) {
        assert(*i);
        int tmp = (*i)->getId();
        assert(tmp > 0);
        max = std::max(max, (unsigned int)tmp);
    }
    max += subthreads.size();
    const unsigned int fork_id = (unsigned int)id->getId();
    std::vector<KEP::KepThreadId*> subthreads = f->getSubThreadIds();
    std::vector<KEP::KepThreadId*>::iterator i;
    for (i=subthreads.begin(); i!=subthreads.end(); i++) {
        assert(*i);
        int tmp = (*i)->getId();
        assert(tmp > 0);
        max = std::max(max, (unsigned int)tmp);
    }
    return (max == fork_id + subthreads.size());
}

void WCRT::visit(std::vector<KEP::KepNode*> n) {
    for (std::vector<KEP::KepNode*>::iterator i=n.begin(); i!=n.end(); i++) {
        assert(*i);
        (*i)->welcome(*this);
    }
}
```

A. C++ Sources
unsigned int WCRT::get_wcrt(void) {
    unsigned int w = 0;
    wcrt_map::iterator i;
    for (i=this->wcrt.inst.begin();i!=this->wcrt.inst.end();i++) {
        w = std::max(w,i->second);
    }
    for (i=this->wcrt.next.begin();i!=this->wcrt.next.end();i++) {
        w = std::max(w,i->second);
    }
    return (w);
}

// create EMIT _TICKLEN,#wcrt statement to be executed at program start
void WCRT::set_ticklen(unsigned int w) {
    assert(this->ckag);
    KEP::Interface *interface = this->ckag->getInterface();
    assert(interface);
    // create integer literal with value 'wcrt'
    KEP::KepType *integer_type = interface->findType("INTEGER");
    assert(integer_type);
    KEP::KepLiteral *wcrt = new KEP::KepLiteral(w,integer_type);
    assert(wcrt);
    // find builtin signal symbol '_TICKLEN'
    KEP::KepSymbol *symbol = interface->findSymbol("_TICKLEN");
    assert(symbol);
    KEP::SignalSymbol *sig_symbol = dynamic_cast<KEP::SignalSymbol*>(symbol);
    assert(sig_symbol);
    // create emit statement
    KEP::KepType *signal_type = interface->findType("SIGNAL");
    assert(signal_type);
    KEP::KepSignal *ticklen = new KEP::ValuedSignal(sig_symbol,signal_type);
    assert(ticklen);
    KEP::Emit *emit = new KEP::Emit(ticklen,wcrt);
    assert(emit);
    // add emit statement to interface statements
    debug->print("add interface statement:",emit);
    interface->addStatement(emit);
}

void WCRT::collect(void) {
    assert(WCRT::ckag);
    KEP::VectorReport *ckag_vector_report = WCRT::ckag->getVectorReport();
    assert(ckag_vector_report);
    KEP::HeaderReport *wcrt_header = new KEP::HeaderReport("WCRT_HEADER","WCRT");
    assert(wcrt_header);
    ckag->addReport(wcrt_header);
}

/*************************************************
*** S E Q U E N T I A L W C R T
*************************************************/
bool WCRT_Sequential::compute(KEP::CKAG *c) {
    debug->print("start WCRT_Sequential::compute:");
    debug->indent();
    wcrt::compute(c);
    std::vector<KEP::KepNode*> n = this->ckag->getNodes();
    for (std::vector<KEP::KepNode*>::iterator i=n.begin();i!=n.end();i++) {
        assert(*i);
        this->compute_inst_wcrt(*i);
        this->compute_next_wcrt(*i);
    }
    assert(wcrt_inst_report);
    std::map<KEP::KepNode*,unsigned int>::iterator i;
    for (i=WCRT::wcrt.inst.begin();i!=WCRT::wcrt.inst.end();i++) {
        assert(i->first);
        KEP::IntegerReport *inst_wcrt = new KEP::IntegerReport("inst",i->second);
        assert(inst_wcrt);
        WCRT::wcrt_inst_report->addReport(i->first,inst_wcrt);
        WCRT::wcrt_inst_report->addReport((i->first)->getStatement(),inst_wcrt);
    }
    // insert collected informations into the wcrt report
    wcrt_report->addReport(wcrt_inst_report);
    wcrt_report->addReport(wcrt_next_report);
    // insert wcrt informations into the ckag graph
    ckag->addReport(wcrt_header);
    }
A. C++ Sources

330 unsigned int w = WCRT::get_wcrt();
    debug->print("wcrt: ",w);
    if (!this->error) {
        WCRT::set_ticklen(w);
    } else {
        WCRT::set_ticklen(0);
    }
    // save wcrt data
    WCRT::collect();
    // create wcrt header
    WCRT::add_wcrt_header();
    debug->unindent();
    debug->print("end WCRT_Sequential::compute.");
    return (!this->error);
}

340 // VISIT TRANSIENT NODE
void WCRT_Sequential::visit(KEP::TransientNode &t) {
    debug->print("start WCRT_Sequential::visit(TransientNode&) ": &t);
    debug->indent();
    KEP::KepStatement *stmt = t.getStatement();
    assert(stmt);
    if (this->mode==INST) {
        debug->print("mode: INST");
        unsigned int w = 0;
        w = std::max(w,this->max_wcrt(t.getChildren()));
        w += WCRT::instruction_cycles(&t);
        debug->print("inst wcrt: ",w);
        WCRT::wcrt.inst[&t] = w;
    } else if (this->mode==NEXT) {
        // do nothing
    } else {
        debug->error("UNDEFINED WCRT MODE");
    }
    debug->unindent();
    debug->print("end WCRT_Sequential::visit(TransientNode&).");
}

350 // VISIT LABEL NODE
void WCRT_Sequential::visit(KEP::LabelNode &l) {
    debug->print("start WCRT_Sequential::visit(LabelNode&) ": &l);
    debug->indent();
    KEP::KepStatement *stmt = l.getStatement();
    assert(stmt);
    if (this->mode==INST) {
        debug->print("mode: INST");
        unsigned int w = 0;
        w = std::max(w,this->max_wcrt(l.getChildren()));
        w += WCRT::instruction_cycles(&l);
        debug->print("inst wcrt: ",w);
        WCRT::wcrt.inst[&l] = w;
    } else if (this->mode==NEXT) {
        // do nothing
    } else {
        debug->error("UNDEFINED WCRT MODE");
    }
    debug->unindent();
    debug->print("end WCRT_Sequential::visit(LabelNode&).");
}

360 // VISIT DELAY NODE
void WCRT_Sequential::visit(KEP::DelayNode &d) {
    debug->print("start WCRT_Sequential::visit(DelayNode&) ": &d);
    debug->indent();
    KEP::KepStatement *stmt = d.getStatement();
    assert(stmt);
    if (this->mode==INST) {
        debug->print("mode: INST");
        unsigned int w = 0;
        w = std::max(w,this->max_wcrt(d.getWabortChildren()));
        w = std::max(w,this->max_wcrt(d.getExitChildren()));
        w += WCRT::instruction_cycles(&d);
        debug->print("inst wcrt: ",w);
        WCRT::wcrt.inst[&d] = w;
    } else if (this->mode==NEXT) {
        debug->print("mode: NEXT");
        unsigned int w = 0;
        w = std::max(w,this->max_wcrt(d.getChildren()));
        w = std::max(w,this->max_wcrt(d.getAbortChildren()));
        w += WCRT::instruction_cycles(&d);
        debug->print("next wcrt: ",w);
        WCRT::wcrt.next[&d] = w;
    } else {
        debug->error("UNDEFINED WCRT MODE");
    }
    debug->unindent();
    debug->print("end WCRT_Sequential::visit(DelayNode&).");
}
7. Appendix

void WCRT_Sequential::visit(KEP::ForkNode &f) {
    debug->print("start WCRT_Sequential::visit(ForkNode&): ", &f);
    debug->indent();
    debug->error("FORK NODES ARE NOT PART OF SEQUENTIAL PROGRAMS!!! ", &f);
    debug->unindent();
    debug->print("end WCRT_Sequential::visit(ForkNode&). ");
}

void WCRT_Sequential::visit(KEP::JoinNode &j) {
    debug->print("start WCRT_Sequential::visit(JoinNode&): ", &j);
    debug->indent();
    debug->error("JOIN NODES ARE NOT PART OF SEQUENTIAL PROGRAMS!!! ", &j);
    debug->unindent();
    debug->print("end WCRT_Sequential::visit(JoinNode&). ");
}

unsigned int WCRT_Sequential::compute_inst_wcrt(KEP::KepNode *n) {
    if (!WCRT::wcrt.inst.count(n)) {
        if (this->error) { // error
            WCRT::wcrt.inst[n] = 0;
        } else if (WCRT::visiting.inst.count(n)) { // cycle
            this->error = true;
            WCRT::wcrt.inst[n] = 0;
            debug->print("cycle during inst wcrt: ", n);
            debug->error("WCRT COMPUTATION NOT POSSIBLE!!! ");
        } else { // computation
            WCRT::visiting.inst.insert(n);
            wcrt_mode tmp = this->mode;
            this->mode = INST;
            n->welcome(*this);
            this->mode = tmp;
        }
    }
    assert(WCRT::wcrt.inst.count(n));
    return (WCRT::wcrt.inst[n]);
}

unsigned int WCRT_Sequential::compute_next_wcrt(KEP::KepNode *n) {
    if (n) {
        if (!WCRT::wcrt.next.count(n)) {
            if (this->error) { // error
                WCRT::wcrt.next[n] = 0;
            } else if (WCRT::visiting.next.count(n)) { // cycle
                this->error = true;
                WCRT::wcrt.next[n] = 0;
                debug->print("cycle during next wcrt: ", n);
                debug->error("WCRT COMPUTATION NOT POSSIBLE!!! ");
            } else { // computation
                WCRT::visiting.next.insert(n);
                wcrt_mode tmp = this->mode;
                this->mode = NEXT;
                n->welcome(*this);
                this->mode = tmp;
            }
        }
        if (WCRT::wcrt.next.count(n)) {
            return (WCRT::wcrt.next[n]);
        } else {
            return (0);
        }
    }
}

unsigned int WCRT_Sequential::max_wcrt(std::vector<KEP::KepNode*> n) {
    unsigned int max = 0;
    for (std::vector<KEP::KepNode*>::iterator i=n.begin(); i!=n.end(); i++) {
        max = std::max(max, this->compute_inst_wcrt(*i));
    }
    return (max);
}

bool WCRT_Parallel::compute(KEP::CKAG *c) {
    WCRT::compute(c);
    // initialisation
    assert(c);
    std::vector<KEP::KepNode*> a = c->getNodes();
    std::vector<KEP::KepNode*>::iterator s;
    for (s=a.begin(); s!=a.end(); s++) {
        assert(*s);
    }
}
A. C++ Sources

// init max_next map
KEP::KepThreadId *id = (*i)->getThreadId();
assert(id);
this->max_inner_next[id] = 0;

// init join_outer_next map
if (KEP::JoinNode *j = dynamic_cast<KEP::JoinNode*>(*i)) {
    this->join_outer_next[j] = 0;
}

// compute all next wcrt's of delay nodes in subthreads
for (i=n.begin();i!=n.end();i++) {
    assert(*i);
    KEP::KepThreadId *id = (*i)->getThreadId();
    assert(id);
    if (dynamic_cast<KEP::DelayNode*>(*i) && id->getId()!=0) {
        this->compute_next_wcrt(*i);
    }
}

// find main thread's id
assert(n.size() > 0);
KEP::KepThreadId *id = (*n.begin())->getThreadId();
assert(id);
id = id->findMainThreadId();
assert(id);

// compute join nodes next wcrt in correct order
id->welcome(*this);

// compute the wcrt for all fork nodes
for (i=n.begin();i!=n.end();i++) {
    if (dynamic_cast<KEP::ForkNode*>(*i)) {
        this->compute_inst_wcrt(*i);
    }
}

// compute the wcrt for all join nodes
for (i=n.begin();i!=n.end();i++) {
    if (dynamic_cast<KEP::JoinNode*>(*i)) {
        // node: this->compute_inst_wcrt does not compute join nodes!!!
        WCRT::Sequential::compute_inst_wcrt(*i);
        WCRT::Sequential::compute_next_wcrt(*i);
    }
}

bool success = WCRT::Sequential::compute(c);

// VISIT TRANSIENT NODE
void WCRT_Parallel::visit(KEP::TransientNode &t) {
    WCRT_Parallel::visit(TransientNode& t);
    debug->unindent();
    debug->print("end WCRT_Parallel::visit(TransientNode&).");
    return (success);
}

// VISIT LABEL NODE
void WCRT_Parallel::visit(KEP::LabelNode &l) {
    WCRT_Parallel::visit(LabelNode& l);
    debug->unindent();
    debug->print("end WCRT_Parallel::visit(LabelNode&).");
}

// VISIT DELAY NODE
void WCRT_Parallel::visit(KEP::DelayNode &d) {
    WCRT_Parallel::visit(DelayNode& d);
    debug->unindent();
    debug->print("end WCRT_Parallel::visit(DelayNode&).");
}

// VISIT TRANSIENT NODE
void WCRT_Parallel::visit(KEP::TransientNode &t) {
unsigned int w = 0;
w = std::max(w,WCRT_Sequential::max_wcrt(d.getChildren()));
w = std::max(w,WCRT::instruction_cycles(&d));
670
debug->print("next wcrt:\t",w);
WCRT::wcrt.next[&d] = w;
}
else {
  debug->error("UNDEFINED WCRT MODE");
  debug->unindent();
  debug->print("end WCRT_Parallel::visit(DelayNode&).\n");
  return;
}
680
// VISIT FORK NODE
void WCRT_Parallel::visit(KEP::ForkNode &f) {
  debug->print("start WCRT_Parallel::visit(ForkNode&):\n","&f);
  debug->indent();
  KEP::Pare *pare = dynamic_cast<KEP::Pare*>(f.getStatement());
  assert(pare);
  KEP::JoinNode *j = f.getJoinNode();
  assert(j);
  KEP::Join *join = dynamic_cast<KEP::Join*>(j->getStatement());
  assert(join);
  KEP::KepThreadId *id = f.getThreadId();
  assert(id);
  if (WCRT_Sequential::mode==INST) {
    debug->print("mode: INST");
    unsigned int w = 0;
    std::map<KEP::Par*,KEP::KepNode*> threads = f.getSubThreads();
    std::map<KEP::Par*,KEP::KepNode*>::iterator t;
    for (t=threads.begin();t!=threads.end();t++) {
      assert(t->second);
      w += this->compute_inst_wcrt(t->second);
    }
    w += WCRT::instruction_cycles(&f);
    debug->print("inst wcrt:\t",w);
    WCRT::wcrt.inst[&f] = w;
  } else if (this->mode==NEXT) {
    // do nothing
    debug->unindent();
    debug->print("end WCRT_Parallel::visit(ForkNode&).\n");
    return;
  } else {
    debug->error("UNDEFINED WCRT MODE");
  }
  debug->unindent();
  debug->print("end WCRT_Parallel::visit(ForkNode&).\n");
  return;
}
690
// VISIT JOIN NODE
void WCRT_Parallel::visit(KEP::JoinNode &j) {
  debug->print("start WCRT_Parallel::visit(JoinNode&):\n","&j);
  debug->indent();
  KEP::Join *join = dynamic_cast<KEP::Join*>(j.getStatement());
  assert(join);
  KEP::ForkNode *f = j.getForkNode();
  assert(f);
  int fork_kind = f->getInstantAttribute();
  std::vector<KEP::KepNode*> exit_children = j.getExitChildren();
  // INST (under construction)
  if (WCRT_Sequential::mode==INST) {
    debug->print("mode: INST\n");
    unsigned int w = 0;
    v = std::max(v,WCRT::instruction_cycles(j));
    if (exit_children.size() > 0) {
      w = std::max(w,WCRT::instruction_cycles(exit_children[0]));
    }
    w += WCRT::instruction_cycles(&j);
    debug->print("inst wcrt:\t",w);
    WCRT::wcrt.inst[&j] = w;
  } else if (WCRT_Sequential::mode==NEXT) {
    // do nothing
  } else {
    debug->error("UNDEFINED WCRT MODE");
  }
  debug->unindent();
  debug->print("end WCRT_Parallel::visit(ForkNode&).\n");
  return;
}
700
// VISIT FORK NODE
void WCRT_Parallel::visit(KEP::ForkNode &f) {
  debug->print("start WCRT_Parallel::visit(ForkNode&):\n","&f);
  debug->indent();
  KEP::Pare *pare = dynamic_cast<KEP::Pare*>(f.getStatement());
  assert(pare);
  KEP::JoinNode *j = f.getJoinNode();
  assert(j);
  KEP::Join *join = dynamic_cast<KEP::Join*>(j->getStatement());
  assert(join);
  KEP::KepThreadId *id = f.getThreadId();
  assert(id);
  if (WCRT_Sequential::mode==INST) {
    debug->print("mode: INST\n");
    unsigned int w = 0;
    std::map<KEP::Par*,KEP::KepNode*> threads = f.getSubThreads();
    std::map<KEP::Par*,KEP::KepNode*>::iterator t;
    for (t=threads.begin();t!=threads.end();t++) {
      assert(t->second);
      w += this->compute_inst_wcrt(t->second);
    }
    w += WCRT::instruction_cycles(&f);
    debug->print("inst wcrt:\t",w);
    WCRT::wcrt.inst[&f] = w;
  } else if (WCRT_Sequential::mode==NEXT) {
    if (fork_kind==KEP::NOT_INSTANTANEOUS || fork_kind==KEP::BOTH) {
      debug->print("mode: NEXT\n");
      unsigned int w = 0;
      std::vector<KEP::KepNode*> c = f->getChildren();
      for (std::vector<KEP::KepNode*>::iterator i=c.begin();i!=c.end();i++) {
        assert(*i);
        KEP::KepThreadId id = (*i)->getThreadId();
        assert(id);
        w += WCRT::instruction_cycles(*i);
      }
    } else if (fork_kind==KEP::INSTANTANEOUS || fork_kind==KEP::BOTH) {
      debug->print("kind: INSTANTANEOUS or BOTH\n");
      w = WCRT::instruction_cycles(&f);
    } else {
      debug->print("kind: NOT_INSTANTANEOUS or BOTH\n");
      w = WCRT::instruction_cycles(&f);
    }
  } else {
    debug->error("UNDEFINED WCRT MODE");
  }
  debug->unindent();
  debug->print("end WCRT_Parallel::visit(ForkNode&).\n");
  return;
}
assert(this->max_inner_next.count(id));
    w += this->max_inner_next[id];
    debug->print("sum of max inner next wcrt's: ", w);
}

w += WCRT_Sequential::compute_inst_wcrt(&j);
debug->print("inner next wcrt: ", w);
WCRT::wcrt.next[j] = w;
}
else {
    debug->print("join is instantaneous: no wcrt next");
    WCRT::visiting.next.erase(&j);
}
else {
    debug->error("UNDEFINED WCRT MODE");
}

debug->unindent();
ddebug->print("end WCRT_Parallel::visit(JoinNode&). ");
}

// compute join next values according to thread hierarchy
void WCRT_Parallel::visit(KEP::KepThreadId &id) {
    debug->print("start WCRT_Parallel::visit(KepThreadId&): ", &id);
    debug->indent();
    std::vector<KEP::ForkNode*> forks = id.getForkNodes();
    for (std::vector<KEP::ForkNode*>::iterator f=forks.begin(); f!=forks.end(); f++) {
        assert(*f);
        std::vector<KEP::KepThreadId*> ids = (*f)->getSubThreadIds();
        std::vector<KEP::KepThreadId*>::iterator j;
        for (j=ids.begin(); j!=ids.end(); j++) {
            assert(*j);
            if ((*j)->getForkNodes().empty()) {
                (*j)->welcome(this);
            }
        }
        assert(*f);
        KEP::JoinNode *j_node = (*f)->getJoinNode();
        assert(j_node);
        this->compute_next_wcrt(j_node);
    }
    debug->unindent();
ddebug->print("end WCRT_Parallel::visit(KepThreadId&). ");
}

// AUXILIARY FUNCTIONS
unsigned int WCRT_Parallel::compute_inst_wcrt(KEP::KepNode *n) {
    unsigned int w = 0;
    if (!dynamic_cast<KEP::JoinNode*>(n)) {
        w = WCRT_Sequential::compute_inst_wcrt(n);
    }
    return (w);
}

unsigned int WCRT_Parallel::compute_next_wcrt(KEP::KepNode *n) {
    assert(n);
    KEP::KepThreadId *id = n->getThreadId();
    assert(id);
    unsigned int w = WCRT_Sequential::compute_next_wcrt(n);
    this->max_inner_next[id] = std::max(w, this->max_inner_next[id]);
    return (w);
}

unsigned int WCRT_Parallel::inner_thread_next_wcrt(KEP::KepNode *n,
std::vector<KEP::KepNode*> nodes) {
    assert(n);
    KEP::KepThreadId *id = n->getThreadId();
    assert(id);
    unsigned int max_inner = 0;
    for (std::vector<KEP::KepNode*>::iterator i=nodes.begin(); i!=nodes.end(); i++) {
        assert(*i);
        KEP::KepThreadId *id_n = (*i)->getThreadId();
        assert(id_n);
        unsigned int w = this->compute_inst_wcrt(*i);
        if (id_n == id) {// inner thread wcrt
            max_inner = std::max(max_inner, w);
        } else {// outer thread wcrt
            KEPS::JoinNode *id_j = id;
            KEPS::JoinNode *j = NULL;
            do {
                assert(id_j->getId() != 0);
                j = id_j->findJoinNode();
                assert(j);
                id_j = j->getThreadId();
                assert(id_j);
            } while (id_j != id_n);
            this->update_outer_next(j, w, n);
        }
    } else {// inner thread wcrt
        max_inner = std::max(max_inner, w);
    }
    return (max_inner);
}
void WCRT_Parallel::update_outer_next(KEP::JoinNode *j, unsigned int w, KEP::KepNode *n) {
    assert(j);
    assert(n);
    assert(this->join_outer_next.count(j));
    KEP::KepThreadId *id = n->getThreadId();
    assert(id);
    // update next wcrt value of a join node
    this->join_outer_next[j] = std::max(w, this->join_outer_next[j]);
    // update outer next statement cycles of particular id
    unsigned int c = 0;
    if (this->join_max_outer_next_cycles.count(j) &&
        this->join_max_outer_next_cycles[j].count(id))
    
    this->join_max_outer_next_cycles[j][id] = std::max(c, WCRT::instruction_cycles(n));
}

unsigned int WCRT_Parallel::get_outer_thread_next(KEP::JoinNode *j) {
    assert(j);
    unsigned int outer_next_cycles = 0;
    KEP::ForkNode *f = j->getForkNode();
    assert(f);
    std::vector<KEP::KepThreadId*> subthread_ids = f->getSubThreadIds();
    const std::vector<KEP::KepThreadId*>::iterator BEGIN = subthread_ids.begin();
    const std::vector<KEP::KepThreadId*>::iterator END = subthread_ids.end();
    std::set<KEP::JoinNode*> join_set;
    if (this->join_max_outer_next_cycles.count(j)) {
        id_map m = this->join_max_outer_next_cycles[j];
        for (id_map::iterator i=m.begin();i!=m.end();i++) {
            assert(i->first);
            outer_next_cycles += i->second;
            // add additional join cycles if needed
            if (std::find(BEGIN, END, i->first) == END) {
                KEP::JoinNode *inner_j = (i->first)->findJoinNode();
                assert(inner_j);
                assert(inner_j != j);
                KEP::KepThreadId *inner_j_id = NULL;
                while (inner_j != j) {
                    if (!join_set.count(inner_j)) {
                        outer_next_cycles += WCRT::instruction_cycles(inner_j);
                        join_set.insert(inner_j);
                    }
                    inner_j_id = inner_j->getThreadId();
                    assert(inner_j_id);
                    inner_j = inner_j_id->findJoinNode();
                    assert(inner_j);
                }
                inner_j_id = inner_j->getThreadId();
                assert(inner_j_id);
                inner_j = inner_j_id->findJoinNode();
                assert(inner_j);
            }
        }
    }
    outer_next_cycles += WCRT::instruction_cycles(j);
    assert(this->join_outer_next.count(j));
    return(this->join_outer_next[j] + outer_next_cycles);
}