WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing

Michael Mendler  Reinhard von Hanxleden and Claus Traulsen
Faculty of Inform. Sys. and Appl. Comp. Sciences  Department of Computer Science
The University of Bamberg  Christian-Albrechts-Universität zu Kiel
michael.mendler@uni-bamberg.de  {rvh|ctr}@informatik.uni-kiel.de

Abstract—The synchronous model of computation together with a suitable execution platform facilitates system-level timing predictability. This paper introduces an algebraic framework for precisely capturing worst case reaction time (WCRT) characteristics for Esterel-style reactive processors with hardware-supported multithreading. This framework provides a formal grounding for the WCRT problem, and allows to improve upon earlier heuristics by accurately and modularly characterizing timing interfaces.

I. INTRODUCTION

Reconciling performance and predictability in embedded systems is a challenge that spans all layers of hard- and software development. However, as observed by Edwards and Lee [1], these abstractions typically limit themselves to encapsulate and guarantee functionality, not timing. Hence, even though there is a significant body of work that addresses timing predictability at different abstraction layers, considering for example schedulability, worst case execution times (WCET), or circuit timing, it is very difficult to transfer results across layers. However, end users are not interested in results that apply only to one layer—they care about timing guarantees for complete systems.

The choice of the model of computation—and its model of time—has a profound influence on how easy or difficult it is to provide timing guarantees across abstraction layers. From the predictability point of view, a very appealing candidate in the embedded systems domain is the synchronous model of computation [2]. Furthermore, languages built on that model generally have a well-established formal semantics that allows reasoning about functional as well as timing properties from the ground up.

In this paper, we first give an overview on how the synchronous model together with a suitable execution platform can provide system-level timing predictability (Section III), and we illustrate this with the case of multi-threaded execution of synchronous programs written in Esterel-like languages [3]. The main contribution of this paper (Section IV) then is the introduction of an algebraic framework for performing Worst Case Reaction Time (WCRT) analyses. These analyses aim to give conservative yet close estimates on the time from capturing inputs to determining outputs, in embedded real-time systems developed with the synchronous model. First experimental results are reported in Section V.

II. RELATED WORK

Most interface models in synchronous programming are restricted to causality issues, i.e., dependency analysis without quantitative time. On the other hand, there exist numerous approaches to classical WCET analysis [5] but only few on WCRT analysis [6], [7].

Logothetis et al. [8] have employed model checking to perform a precise timing analysis for the synchronous language Quartz, which is similar to Esterel. However, their problem is WCET since they are interested in computing the number of logical ticks required to perform a certain transformational computation, such as a primality test.

The modules of André et al. [9] do not permit instantaneous interaction. Such a model is not suitable for WCRT. Hainque et al. [10] use a topological abstraction of the underlying circuit graphs (or syntactic structure of Boolean equations) to derive a fairly rigid component dependency model with the effect that multi-threaded execution cannot be modeled compositionally. The interface model also does not cover data dependencies and thus cannot deal with dynamic schedules and does not support WCRT, either.

The causality interfaces of Lee et al. [11] are more flexible. These are functions associating with every pair of input and output ports an element of a dependency domain D, which expresses if and how an output depends on some input. Causality analysis is then performed by multiplication on the global system matrix. Using an appropriate dioid structure D, one can perform the analyses of Hainque et al. [10] as well as restricted forms of WCRT. However, Lee’s interfaces cannot express the difference between an output depending on the joint presence of several values as opposed to depending on each input individually. Thus they do not support full AND- and OR-type synchronization dependencies and hence cannot represent neither multi-threading nor multi-processing. The work reported here can be seen as an extension of [11] to address these deficiencies.

Similar restrictions apply to recent work [12], [13] combining network calculus [14], [15] with real-time interfaces. These
works are concerned with the compositional modeling of regular execution patterns rather than stabilization processes inside each execution cycle of a synchronous program. Existing interface theories [11], [12], [13], which aim at the verification of resource constraints for real-time scheduling, handle timing properties such as task execution latency, arrival rates, resource utilization, throughput, accumulated cost of context switches, and so on. However, the dependency on data and control flow is largely abstracted. For instance, since the task sequences of Henzinger and Matic [13] are independent of each other, their interfaces do not model concurrent forking and joining of threads. The causality expressible there is even more restricted than that by Lee et al. [11] in that it permits only one-to-one associations of inputs with outputs. The interfaces of Wandeler and Thiele [12] for modular performance analysis in real-time calculus are like those of Henzinger and Matic [13] but without sequential composition of tasks and thus do not model control flow as we do here.

In so far as WCRT analysis aims to obtain exact bounds on the duration of stabilization processes with synchronous feedback, it is related to the timing analysis of combinational circuits (see, e.g., [16], [17], [18], [19]) which is known to be NP-complete. Although WCRT analysis for single or multi-threaded synchronous processing can mostly be performed in max-plus as opposed to min-max-plus algebra, the inherent data dependency still makes it computationally intractable without sophisticated heuristics. The work presented here fits into a general and expressive interface theory [20] for stabilization processes which has been developed to provide a semantical foundation for such heuristics. It supports modularization and hierarchical abstraction and systematizes earlier work on combinational timing analysis.

III. SYNCHRONICITY AND TIMING PREDICTABILITY

The synchronous model of computation divides physical time into a sequence of discrete ticks, or instants. The abstraction is that at each tick, outputs are synchronous with the inputs. In other words, computations take place instantaneously, interspersed with durations of inactivity between ticks. Synchronous languages generally do not permit unbound computations within a tick. For example, the language Esterel provides a loop construct [3], however, instantaneous loops are forbidden; i.e., each loop iteration must include at least one tick-delimiting instruction, and the compiler must be able to verify this. This simplifies the problem of determining the maximal number of instructions per tick, which leads to the worst case reaction time (WCRT). The situation is quite different for imperative languages such as Java or C, which permit unbounded loops and unbounded recursion, and thus only language subsets (e.g., with statically bounded loop iterations) are amenable to WCET analyses. Another helpful characteristic of (strict) synchrony is that the statuses of signals, which are basic communication means in synchronous programs, evolve monotonically. There can be no oscillations between signal presence and absence, thus guaranteeing convergence after a finite number of computations. This contrasts, for example, with Harel’s original Statecharts dialect [21], which assumes a weaker form of synchrony in which computations are also assumed to not consume any time, but signal statuses are allowed to oscillate and computations within a tick are unbounded. Finally, the synchronous paradigm also supports concurrent and preemptive control flow, with a deterministic semantics regarding both functionality and timing characteristics. This again contrasts with classical imperative languages, which either do not support non-sequential control flow at all (e.g., C relegates this to the OS level, subject to run-time scheduling decisions), or support it only in a rather haphazard fashion (e.g., Java threads [22]).

Synchronous programs may be compiled into hardware or software. The traditional software design flow is to first compile the synchronous program into a classical imperative language, such as C, and to run the resulting program on a standard micro processor [23]. This approach preserves the nice semantical properties of synchronous programs at a functional level. The timing properties, however, are only partially preserved with this approach. Computations are still finite per tick, and the synthesized C-code has no unbounded loops, for example. However, depending on the synthesis approach used, the control flow may still be rather complex and difficult to analyze (e.g., computed gotos). Furthermore, standard processors typically employ various techniques that improve average execution time, at the expense of worst case execution time and predictability [24].

An alternative, more recent approach for executing synchronous programs is to run them on processors that directly support reactive control flow. This reactive processing approach builds on instruction set architectures (ISAs) that can express concurrency and preemption and preserve functional determinism [25]. Note that this approach does typically not strive for high-performance processing that maximizes average execution times and uses hardware acceleration techniques such as caching and pipelining [1]. Instead, the focus here is on predictable architectures with fixed instruction cycle times.

There have been various proposals on how to support concurrency in reactive processing, including sequentialization, parallel execution, and, most recently, multi-threading [26], [27], [28]. The latter one appears to be the most effective at this point, and significantly outperforms classical software-based execution strategies while using minimal resources.

In the multi-threaded reactive processing approach, timing determinism is assured by a combination of static scheduling, hardware-supported context switching, and fixed machine instruction execution times. This has been exploited in a compiler which translates Esterel programs into multi-threaded assembler code for the Kiel Esterel Processor (KEP), and as part of the compilation process analyzes the WCRT in terms of instruction cycles [7].

The WCRT analysis technique developed by Boldt et al. [7] already provide fairly promising results with a reported accuracy typically in the 30–40% range. However, this heuristics still makes conservative and simplifying assumptions and is not grounded in a formal timing model. To illustrate this,
consider the small program $G$ in Fig. 1(a) and the corresponding assembler (b). Considering that each of the basic instructions present, emit, goto takes 1 instruction cycle (ic) regardless how it is entered or exited, the longest-path heuristic implemented in the KEP compiler [7] will compute a WCRT of 6 ics. This, however, is overly conservative, as the longest path makes contradictory assumptions (signal $l$ present and absent at the same time). Furthermore, the existing WCRT algorithms are neither compositional nor scalable in terms of precision. They are global analyses on the complete and fully-expanded control-flow graph of a monolithic program and run at the ground level of atomic program statements rather than hierarchical sub-systems.

In this paper we propose a theory of WCRT interfaces for synchronous programming and show how it can be employed to obtain type-directed and modular WCRT analyses which (1) give precise statements about exactness and coverage of timing values, supporting a variety of timing abstractions, (2) are dedicated to express the imperative synchronous programming languages, and (3) are scalable across component hierarchies and the software-hardware abstraction boundary. As an interface theory our WCRT algebra operates on matrices of delay values characterizing whole sub-sytems rather than individual nodes like existing graph-theoretic WCRT algorithms do. Like the propositional stabilization theory presented in [20] it combines max-plus algebra ($\mathbb{N}, \max, +, 0, -\infty$) [14] with an intuitionistic refinement of Boolean logic to reason about implicit control-flow.

IV. THE WCRT ALGEBRA

An execution $\sigma$ is a finite and monotonically increasing sequence of sets of control signals $A \in S$, which can be data-signals or control-flow labels. We also use activation controls active($v$) $\in S$ for nodes $v$ in the hierarchical decomposition of the program. An execution $\sigma$ models the micro-sequence of instruction cycles (ic) which are executed by a given thread within a single synchronous instant. Each step $\sigma(i) \leadsto \sigma(i+1)$ records the change of controls between two successive activations of the thread. The empty execution $\sigma = \emptyset$ is included as a degenerated case. The difference $\Delta_t = \sigma(i+1) \setminus \sigma(i)$ may be an arbitrary subset of $S$. It will encompass more than one signal when the thread forks into concurrent sub-threads or if other concurrent threads get executed between the two activations $i$ and $i+1$ of the thread represented by $\sigma$. 

A. Scheduling Types

A set of executions $S$ defines a schedule. The possible schedules of a program are specified by a scheduling type

$$\phi ::= A \mid true \mid false \mid \phi \land \phi \mid \neg \phi \mid \phi \lor \phi \mid \phi \parallel \phi \mid \phi \circ \phi$$

We write $S \models \phi$ ($\sigma \models \phi$) to say that schedule $S$ (execution $\sigma$) validates the type $\phi$. As a type, each signal $A \in S$ represents the statement that “$A$ is active (is present, traversed, scheduled) in all executions of the schedule.” The constant true is validated by all schedules and false only by the empty execution or the schedule which contains the empty execution only. The type operators $\neg$, $\circ$ are (intuitionistic) negation and implication. The operators $\lor$ and $\land$ are two forms of logical disjunction to encode internal and external non-determinism and $\parallel$, $\circ$ are two forms of logical conjunction related to true concurrency (multi-processing) and interleaving concurrency (multi-threading), respectively. Finally, $\phi$ is the operator to express execution delays. This type syntax permits definitions provided in the following.

A basic control type is an expression $\zeta$ built from literals $A$, $\neg A$ ($A \in S$) and constants true, false using conjunction $\land$ and disjunction $\lor$. Basic control types satisfy $S \models \zeta$ iff $\sigma \models \zeta$ for all $\sigma \in S$, i.e., they express properties of individual executions. On executions, $\zeta$ behaves like a standard Boolean combination of the atomic statements $A$ (“$A$ present throughout”) and $\neg A$ (“$A$ absent throughout”). For instance, $\sigma \models A \lor \neg A$ says that signal $A$ is constant in $\sigma$, i.e., it is either present from the start, or never becomes active. Since signals which are not active initially may occur in the course of an execution, the type $A \lor \neg A$ is not a tautology, i.e., $A \lor \neg A \not\models true$. This intuitionistic nature of negation is crucial to handle the semantics of synchronous languages in a compositional and fully abstract way [29]. For special signals like the activation of nodes active($T$) it is safe to assume $active(T) \lor \neg active(T) \models true$ since these state signals are decided at the start of every instant. Every basic control has an equivalent disjunctive normal form $\zeta = \bigoplus_i \bigwedge_j l_{ij}$ over literals $l_{ij}$. Basic controls $\zeta$ are used to specify scheduling interaction at the input and output side of a program node. When used as an output we need to express that $\zeta$ occurs delayed after some maximal number of ics, $d$ say. We write $\sigma \models d : \zeta$ to abbreviate of $\sigma' \models \zeta$ where $\sigma' = \sigma(d) \sigma(d+1) \ldots \sigma(|\sigma| - 1)$ is the suffix of $\sigma$ starting after $d$ ics. Note that if the delay is larger than the length of the execution, $d > |\sigma| - 1$, then this suffix is empty $\sigma' = \emptyset$ and thus $\sigma \models d : \zeta$ for all $\zeta$, even $\zeta = false$ is validated. This is natural since by stepping beyond the final event within a thread’s instant an inconsistent
reactive box with input and output controls. The paths inside seen in Fig. 2 illustrate the four ways in which a reactive node may (a) pass straight through the node entering at some input T specifies a set of executions. To capture the bounds, we associate with each interface type a delay matrix of shape D. An input control is an expression \( \psi = \circ \xi_1 \oplus \circ \xi_2 \oplus \cdots \oplus \circ \xi_n \) with basic controls \( \xi_j \). \( S \models \psi \) specifies that schedule \( S \) reaches at least one of the controls \( \xi_j \) after a bounded number of instruction cycles (ics). The selection \( \oplus \) of which \( \xi_j \) is activated is an internal choice which is dynamically resolved during each execution. Each operator \( \circ \) stands for a possibly different delay depending on which output \( \xi_j \) is taken. In contrast to this, an output control such as \( \psi = \circ (\xi_1 \oplus \xi_2 \cdots \oplus \xi_n) \) only specifies a single bound for all exits \( \xi_j \).

An input control is an expression \( \phi = \xi_1 \lor \xi_2 \lor \cdots \lor \xi_m \) where the disjunction \( \lor \) refers to the external non-determinism resolved by the environment which determines how a program node is started. There is also no delay involved which is why we do not need operator \( \circ \). Formally, \( S \models \phi \) if there is at least one \( \xi_i \) such that \( S \models \xi_i \).

Notice the change of quantifiers between input and output controls regarding executions: \( S \models \xi_i \lor \xi_j \) requires \( \exists i \in \{1,2\} \forall \sigma \in S, \sigma \models \xi_i \) which is an external choice, whereas \( S \models \xi_1 \lor \xi_2 \lor \xi_j \) is \( \forall \sigma \in S, \exists i \in \{1,2\}, \sigma \models \xi_i \) which expresses an internal choice.

B. Interface Types

We build interface types for program fragments as implications \( \phi \models \psi \) between input controls \( \phi = \bigvee_{i=1}^{n} \xi_i \) and output controls \( \psi = \bigoplus_{j=1}^{m} \circ \xi_j \). The input controls \( \phi \) capture all the possible ways in which the program fragment can be started within an instant and the output controls sum up the ways in which it can be exited during the instant. Intuitively, \( S \models \phi \models \psi \) says that whenever any set of executions from schedule \( S \) enters the program through one of the input controls \( \xi_i \), then within some bounded number \( d_{ij} \) of ics all these executions are guaranteed to exit through one of the output controls \( \xi_j \). The bounds \( d_{ij} \) may depend on the choice of input and output control, in general. To capture the bounds, we associate with each interface type a delay matrix of shape \( n \times m \). Our type specifications then become logical expressions of the form \( D : \phi \models \psi \) consisting of a timing matrix \( D \) together with an interface type \( \phi \models \psi \). The former describes the quantitative aspect of scheduling, the latter captures the qualitative part of the interface. Formally, \( \phi \models \psi \) is a type specification for schedules \( S \) and the instrumented \( D : \phi \models \psi \) specifies a set of executions.

Fig. 2 depicts a program fragment \( T \) abstracted into a reactive box with input and output controls. The paths inside \( T \) seen in Fig. 2 illustrate the four ways in which a reactive node \( T \) may participate in the execution of a logical tick: Threads may (a) pass straight through the node entering at some input control \( \zeta \) and exiting at output control \( \xi \); (b) enter through \( \zeta \) but pausing inside, waiting there for the next instant; (c) start the tick inside the node and eventually (instantaneously) leave through some exit control \( \xi \), or (d) start inside the node and never leave it during the current instant. These paths or rather sections of a path are called through paths, sink paths, source paths and internal paths, respectively.

The interface type for such a node \( T \) (considering only one input control \( \zeta \) and one output control \( \xi \)) separates these different paths and associated WCRT values:

\[
T = \left( \begin{array}{cc}
\frac{d_{thr}}{d_{src}} & \frac{d_{src}}{d_{int}} \\
\frac{d_{sink}}{d_{int}} & \frac{d_{int}}{\infty}
\end{array} \right) \models (\xi \lor active) \lor (\circ \xi \lor \circ wait)
\]

If one of the paths does not exist its associated delay is set to \(-\infty\). A node \( T \) can be classified according to the paths that are executable in it. We define the (not necessarily disjoint) sets of through nodes, \( N_{thr} = \{ T \mid d_{thr} \geq 0 \} \), source nodes, \( N_{src} = \{ T \mid d_{src} \geq 0 \} \), sink nodes, \( N_{sink} = \{ T \mid d_{sink} \geq 0 \} \), and internal nodes, \( N_{int} = \{ T \mid d_{int} \geq 0 \} \). A delay node is a node with at least one non-instantaneous path (\( N_{del} = N_{src} \cup N_{sink} \cup N_{int} \)). A strong delay node is a delay node without any through path (\( N_{del} = N_{del} \setminus N_{thr} \)). A transient node is a through node that contains only through paths, i.e., \( d_{src} = d_{sink} = d_{int} = \infty \) (\( N_{trans} = N_{thr} \setminus N_{del} \)). Each cyclic dependency loop in the program must be broken by at least one strong delay node, which corresponds to the rule mentioned earlier that forbids instantaneous loops.

In general, the interface type of a program \( T \) will mention a number of controls \( \xi_1, \xi_2, \ldots, \xi_m \) and \( \xi_1, \xi_2, \ldots, \xi_n \) on the input and output side for which the type would be

\[
T = D : (\xi_1 \lor \xi_2 \cdots \lor \xi_m) \lor (\circ \xi_1 \lor \circ \xi_2 \cdots \lor \circ \xi_n) \quad (1)
\]

with a WCRT matrix \( D \) of shape \( n \times m \). A composite program will be made up of a number of program fragments \( T_i \) each with its interface \( D_i : \phi_i \models \psi_i \). The total specification is the logical conjunction \( \bigwedge_i D_i : \phi_i \models \psi_i \) in WCRT type algebra. The basic controls appearing in \( \phi_i, \psi_i \) describe the causal dependencies between the nodes \( T_i \). In its general form, WCRT analysis amounts to a transformation

\[
\bigwedge_i D_i : \phi_i \models \psi_i \preceq D : \phi \models \psi \quad (2)
\]

in which the individual timing interfaces \( D_i \) are combined into a total delay matrix \( D \) for an external interface \( \phi \models \psi \) such
that \( D \) is the smallest (component-wise) matrix of values such that (2) holds. The external interface \( \phi \triangleright \psi \) determines the functional precision with which we are computing the WCRT of a composite system. For instance, instead of an interface like (1), which distinguishes \( m \) input and \( n \) output controls, a less discriminative type \( \zeta \triangleright \omega \) with \( \zeta = d \bigvee_{j \in J} G_j \) and \( \xi = d \bigcirc \bigoplus_{j \in J} E_j \) might consider merely subsets \( I \subseteq \{1, \ldots, m\} \) and \( J \subseteq \{1, \ldots, n\} \) of inputs and outputs bundled into a single control. Such an interface \( \zeta \triangleright \omega \), which specifies only one delay value is more abstract than (1). Of course, we do not expect to get an equivalence \( \cong \) but only an inclusion \( \subseteq \) in (2) if the calculation of \( D \) involves timing abstractions. We can trade off precision and efficiency of the WCRT analysis within wide margins by choosing different types \( \phi_i \triangleright \psi_i \) for the components and \( \phi \triangleright \psi \) for the composite program in (2). By logical transformations of interfaces, various optimizations can be achieved including such as those employed by classic combinational timing analyses [20].

C. An Example

To illustrate the use of WCRT types consider again Fig. 1. Each node \( v_1 \rightarrow v_7 \) in the control-flow graph (c) of the associated Esterel program (a) is compiled into an assembler instruction (b) which is entered either sequentially through its instruction number L4–L10 or through an explicit jump to a control flow label such as G0–G3. For instance, node \( v_7 \) is accessed both through its linear instruction number L6 as well as by jump to its label G1. In contrast, node \( v_4 \) is only accessed through its line number L7 while node \( v_5 \) only by jumping to its label G3. The present nodes \( v_1 \) and \( v_3 \) are tests which branch to their two successor instructions depending on the status of signal \( I \). If \( I \) is present then \( v_1 \) moves to instruction \( v_2 \) which immediately follows it and if \( I \) is absent then \( v_1 \) passes control to instruction \( v_3 \) by jumping to label G1.

An interface which only considers the input G0 and computes the longest path through G is (6) : \( G0 \triangleright o L11 \). A full WCRT specification encapsulating program G as a component would require mention of program labels G1, G3, G2 which are accessible from outside for jump statements. Therefore, the interface type of G would be (6, 4, 3, 1) : \((G0 \lor G1 \lor G3 \lor G2) \supset o L11 \). This is still not the most exact description of G since it does not express the dependency of the WCRT on signal \( I \). In particular, the longest path of length 6 from G0 to L11 is not executable. To capture this we consider signal \( I \) as just another control input and refine the WCRT scheduling type of G as follows: (5, 3, 3, 3, 1) : \((G0 \land I) \lor (G0 \land \neg I) \lor (G1 \land I) \lor (G1 \land \neg I) \lor (G3 \lor G2) \supset o L11 \). The inclusion of signal \( I \) in the interface has now resulted in the distinction of two different delays 3 and 4 for \( G1 \supset o L11 \) depending on whether \( I \) is present or absent during the reaction. On the other hand, G0 split into controls \( G0 \land I \) and \( G0 \land \neg I \) produces the same delay of 5 \( \mu \) in both cases, which is a decrease of WCRT compared to 6 from above. Assuming that input signal \( I \) is causally stable, i.e., \( I \oplus \neg I \cong true \), the two entries of value 5 can be merged into a single value as in (5, 3, 3, 3, 1) : \((G0 \lor (G1 \land I) \lor (G1 \land \neg I) \lor (G3 \lor G2) \supset o L11 \). In the same vein, we could further bundle G1\land I \) and G3 into a single input control \((G1 \land I) \lor G3 \) with delay 3. This finally gives (5, 3, 4, 1) : \((G0 \lor ((G1 \land I) \lor G3) \lor (G1 \land \neg I) \lor G2) \supset o L11 \). Still, if we only ever intend to use \( G \) as a composite node from G0 to L11, the typing (5) : \( G0 \triangleright o L11 \), which takes care of signal dependency on I, might be sufficient.

All operations on interfaces and WCRT analyses are supported by semantically sound transformation rules in WCRT type algebra. The logical manipulation of types often can be done implicitly and hard-coded into the graph-theoretic search strategies that make up the cleverness of a particular WCRT algorithm. Where interface types are not used directly in the calculations they provide for a highly compositional fine-analysis which allows us to validate WCRT algorithms in terms of precise statements about correctness and exactness. Due to their logical-symbolic nature WCRT interfaces can be applied in rather general situations which involve data and higher control-flow constructs as used in synchronous programming.

V. Results

To evaluate our approach, we prototypically implemented some of the key ideas. We identify blocks with threads and compute the through, source, sink and internal WCRT for each thread independently. All outgoing transitions from a thread are abstracted into one. The results for some test-cases, taken from the Estbench test suite [30], can be found in Table I. Since the approach does not consider traps yet, we had to replace traps by local signals and weak abortion. This is trivial for these examples. In general, the transformation can be done analogously to the hardware synthesis from Esterel [23].

<table>
<thead>
<tr>
<th>Module name</th>
<th>WCRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>abro</td>
<td>11</td>
</tr>
<tr>
<td>atds</td>
<td>11</td>
</tr>
<tr>
<td>mca200</td>
<td>60</td>
</tr>
<tr>
<td>runner</td>
<td>34</td>
</tr>
<tr>
<td>tcint</td>
<td>1779</td>
</tr>
<tr>
<td>watch</td>
<td>1872</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This limited implementation already leads to improvements over earlier analyses [7] in most of the tested example cases. Still, the analysis is not as exact as it could be. In two cases (mca200 and watch) we are slightly worse than the graph based approach, because the interface approach so far does not distinguish between immediate and delayed aborts. The implementation could also be improved, e.g., by unbundling outgoing thread transitions and other heuristics. The theory could be further strengthened, e.g., by directly integrating abortion in the control flow graph.
VI. CONCLUSION AND FUTURE WORK

We introduced an interface algebra for compositional analysis of WCRT in synchronous multi-threading and illustrated this with a small, sequential example. The full report [4] expands on this in several areas, notably on the handling of delay nodes and concurrency, and on how to trade off precision against efficiency by interface bundling.

This algebraic approach is very flexible: from considering all possible data, which gives an exact WCRT for the price of possible exponential computation time, to abstracting from all internal behavior, which is very fast but might lead to a large over-approximation, all levels of exactness can be applied. Beside the handling of control data, the more systematic treatment of parallel execution leads to tighter WCRTs. Since the interfaces are compositional, we should also be able to get a better performance for the WCRT computation on large programs. Furthermore, data-dependencies with arbitrary precision can be easily expressed in the interface algebra, to rule out impossible executions and get an even tighter WCRT.

The flexibility for modularization and abstraction together with the tight semantic coupling of numeric and functional information are the main advantages over previous approaches on Boolean timing analysis [19].

Encouraged by our experimental results, we want to fully implement a suite of WCRT algorithms based on the new interfaces. We would like to extend our interfaces to cover exits and non-immediate aborts as well. A further step would be to integrate thread priorities into the interfaces, to reduce the number of considered paths. At the moment, aborts are handled by adding transitions to all pause nodes inside them. It might be more natural to extend the control flow graphs by hierarchy to directly express the hierarchical nature of aborts. This should easily be captured by our interfaces. A central part of future work on the hardware side will consist in developing a formal operational execution model of KEP and verifying our WCRT algorithms with respect to this model.

REFERENCES