This is the author prepared accepted version. © 2019 IEEE.
S. Natarajan, M. Nasri, D. Broman, B. B. Brandenburg and G. Nelissen. From Code to Weakly Hard Constraints: A Pragmatic End-to-End Toolchain for Timed C.
In the Proceedings of IEEE Real-Time Systems Symposium (RTSS), Hong Kong, China, 2019.
DOI: http://dx.doi.org/10.1109/RTSS46320.2019.00025

# From Code to Weakly Hard Constraints: A Pragmatic End-to-End Toolchain for Timed C

Saranya Natarajan*, Mitra Nasri⋆, David Broman*, Björn B. Brandenburg†, and Geoffrey Nelissen‡
*KTH Royal Institute of Technology, Sweden ⋆Delft University of Technology, Netherlands
†Max Planck Institute for Software Systems (MPI-SWS), Germany
‡CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Portugal

***Abstract*—Complex real-time systems are traditionally developed in several disjoint steps: (i) decomposition of applications into sets of recurrent tasks, (ii) worst-case execution time estimation, and (iii) schedulability analysis. Each step is already in itself complex and error-prone, and the composition of all three poses a nontrivial integration problem. In particular, it is challenging to obtain an end-to-end analysis of timing properties of the whole system due to practical differences between the interfaces of tools for extracting task models, execution time analysis, and schedulability tests. To address this problem, we propose a seamless and pragmatic end-to-end compilation and timing analysis toolchain, where source programs are written in a real-time extension of C, called Timed C. The toolchain automatically translates timing primitives into executable code, measures execution times, and verifies temporal correctness using an extended schedulability test for non-preemptive generalized multiframe task sets. Novel aspects of our approach are: (i) both soft and firm tasks can be expressed at the programming language level and stated timing requirements are automatically verified by the schedulability test, and (ii) the schedulability test outputs per-job response-time information that enables a new approach to sensitivity analysis. Specifically, we perform a weakly hard sensitivity analysis that determines the worst-case execution time margins for the strongest still-satisfied $(M, \mathcal{K})$ constraint, where $M = m_1 + \ldots + m_N$ denotes the number of deadline misses across the entire task set, and $\mathcal{K} = \{k_1, \ldots, k_N\}$ is the set of windows of interest of the different tasks. The toolchain is implemented as a source-to-source compiler, freely available as open source, and conveniently distributed as a Docker container.**

## I. Introduction

The implementation and validation of a real-time system involves a number of distinct steps. First, the application is decomposed into a set of recurrent tasks. The properties and constraints of the tasks (such as periodicities, dependencies, and deadlines) are either specified natively in a real-time programming language (such as Ada [12], a real-time extension of C [17, 31], or RTSJ [44]), or by using system calls provided by a real-time operating system (RTOS). In a second step, the *worst-case execution time* (WCET) [45] of each task is determined, either using static analysis methods that determine a conservative, safe upper bound on the actual WCET [16, 19], or by using measurement-based methods [6] that estimate a WCET bound based on a set of execution traces. These WCET estimates together with task properties such as their periodicity are then given as input to a schedulability analysis.

The viability of this traditional design methodology, however, is problematic from both practical and theoretical viewpoints. Practically speaking, research on tools and methods for the different steps proceeds largerly in isolation, making it challenging to integrate individual solutions into an automatic toolchain. Theoretically, the soundness of each step relies on the correctness of previous steps. For instance, a schedulability analysis of hard real-time systems requires the existence of a safe WCET bound for each task. This is, however, a very strong assumption: on the one hand, static methods are safe and conservative [16, 19, 45], but rely on the correctness of the timing model of the hardware, which is hard to obtain for complex modern architectures. Measurement-based methods are on the other hand inherently unsafe because they can only provide observations for a subset of all possible execution traces [45]. Thus, in practice, uncertain WCET estimates, together with timing imprecisions induced by the RTOS or underlying hardware platform, render even a theoretically "hard" schedulability result less than 100% certain. A binary analysis outcome ("schedulable" or "not schedulable") is clearly unsatisfying when the inputs have a high level of uncertainty.

To make progress on both fronts—tool integration and platform unpredictability—we propose an *end-to-end toolchain for Timed C*, a recently proposed dialect of the C programming language with explicit timed semantics [31]. As illustrated in Fig. 1, our toolchain seamlessly integrates **(i)** a source-to-source compiler [31], **(ii)** automatic instrumentation for measurement-based timing analysis (Section III), **(iii)** a uniprocessor *schedulability analysis* extended with support for Timed C semantics (Section IV), and **(iv)** a novel *sensitivity analysis* for *weakly hard constraints* [18] (Section V), a classic way of specifying real-time requirements with a *quantifiable* degree of "softness."

The proposed toolchain targets non-preemptively scheduled uniprocessors because this initial version is primarily aimed at microcontrollers and similarly small, embedded platforms. It is convenient for programmers since it automatically translates Timed C's native timing primitives into executable code, measures execution times, and assesses the application's temporal correctness with a novel weakly hard sensitivity analysis based on an extended uniprocessor schedulability test [30] for non-preemptive generalized multiframe tasks [5] *without exposing any of the details pertaining to the integration of the individual methodologies and tools to the programmer*.

Furthermore, our toolchain is *pragmatic* in two major ways. First, it works for any hardware platform supported by a C compiler (since the Timed C source-to-source compiler

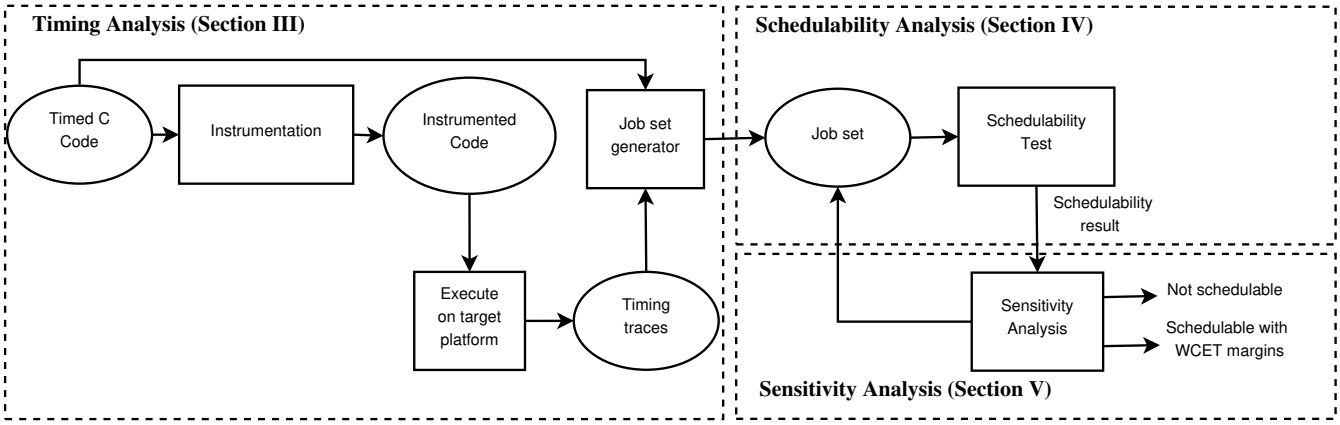



Fig. 1: Overview of the proposed end-to-end toolchain for Timed C. The overall methodology and toolchain consists of three main parts: a measurement-based WCET analysis that automatically instruments and profiles the real-time program on the target platform (Section III), an exact schedualbility analysis (Section IV), followed by a sensitivity analysis (Section V), which iteratively searches for boundaries in the space of task set parameters for which the system exhibits a varying number of deadline misses.

translates to "plain" C) that allows for dynamic execution-time measurements (since we do not rely on static WCET analysis), which makes it readily applicable in many contexts and with a low barrier to entry. Second, our toolchain embraces the inherent platform unpredictability that realistically cannot be avoided on contemporary commodity hardware by reporting *margins instead of guarantees*. Specifically, instead of providing a simplistic "yes/no" result, the toolchain performs a weakly hard sensitivity analysis that determines the strongest still-satisfied weakly hard real-time constraint (across the entire task set) as a function of increasing maximum execution times.

For example, after compiling a given task set and and profiling it on the deployment platform, the toolchain might report that **(i)** no deadlines will be missed assuming the observed WCET estimates, **(ii)** one deadline miss cannot be ruled out if WCETs increase by, say, 30%, **(iii)** two deadline misses cannot be ruled out if WCETs increase by 43%, **(iv)** three deadline misses cannot be ruled out if WCETs increase by 61%, and so on. Pragmatically speaking, such output is much more useful to an engineer than a simple "yes/no" schedulability result, or a single response-time bound based on uncertain WCET estimates, because it provides insight into the system's *robustness*, i.e., a quantitative assessment of the margin of error in the reported timing properties. Especially given that many embedded real-time systems can typically tolerate "a few" deadline misses (e.g., this is true for most control systems), it can be highly valuable to learn that an unexpected WCET increase by $X\%$ will result in no more than $Y$ deadline misses (across a configurable window of $k_i$ consecutive invocations for each task $\tau_i$).

In addition to the practical aspects of the proposed toolchain—which is freely available[1] as an open-source project and distributed as a Docker container—this paper makes two algorithmic contributions to the state of the art. First, we provide a *uniprocessor schedulability test* for Timed C programs that realize a set of periodic *generalized multiframe tasks* (GMF) [5] scheduled by a non-preemptive job-level fixed-priority (JLFP) scheduling algorithm, which is obtained by extending a recent schedulability test [30] for independent non-preemptive jobs to support precedence constraints and the forced abortion of jobs (which is needed for a key feature of Timed C, namely "firm timing points," as discussed in Section II-A).

Second, we provide the *first sensitivity analysis* for weakly hard real-time systems, which yields a WCET margin for the strongest still-satisfied $(M, \mathcal{K})$ specification, that is, the largest factor by which all WCET estimates can be scaled while missing at most $M = m_1 + \ldots + m_N$ deadlines across all $N$ tasks w.r.t. a set $\mathcal{K} = \{k_1, \ldots, k_N\}$ of user-configurable windows of interest of $k_i$ jobs each.

## II. Background and System Model

This section provides a brief introduction to Timed C, its key features (Section II-A), and then defines a system model to represent a wide class of Timed C programs (Section II-B).

### A. The Timed C Language

Timed C [31] is a recently introduced programming language that is designed to expose fine-grained control of program timing to application programmers. In particular, it offers a set of temporal and concurrent constructs with a clear temporal semantics at the language level that enable programmers to easily detect and react to transient overruns at the granularity of individual blocks. For instance, Timed C provides a safe construct for interrupting and aborting the execution of a given code fragment when, for any reason, it does not finish by its deadline. These features empower the programmer to have precise control of the timing of I/O interactions, making Timed C a good alternative for embedded real-time systems.

Timed C uses the concept of *timing points* to let the programmer express and combine various timing constraints as first-class constructs. The current implementation of the Timed C language provides programmers with two types of timing point primitives: **(i)** soft timing points and **(ii)** firm timing points. A *soft timing point* (STP) is specified as

```
stp(expr1, expr2, n)
```

[1] https://github.com/timed-c/end-to-end-toolchain

```
1 task foo(){
2   stp(20,inf,ms);
3   while(1){
4     work();
5     stp(60,40,ms);
6   }
7 }
```

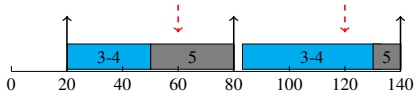



Fig. 2: A Timed C program realizing a simple periodic task and a timing diagram of its execution. The task has offset 20, period 60, and relative deadline 40 ms. In the timing diagram, the upward arrows show arrival times and the dashed downward arrows show deadlines.

where the arguments `expr1` $\in \mathbb{N}$ and `expr2` $\in \mathbb{N}$ are the lower bound and the upper bound on the amount of delay relative to the *previous* timing point, respectively. The argument `n` $\in \mathbb{N}$ is the resolution exponent of the time values, i.e., the resolution is $10^{-n}$ seconds (a common choice is the macro `ms`, which expands to 3 to yield millisecond resolution). Equivalently, `expr1` can be seen as the relative arrival time of the code fragment following the timing point with respect to the arrival time of the code fragment preceding the current timing point, while `expr2` can be seen as the relative deadline of the code fragment preceding the current timing point.

Fig. 2 shows a Timed C program example with two soft timing points, together with its timing trace. The program implements a simple periodic task with period 60, an offset 20, and a soft deadline of 40 ms. The offset is applied using the first `stp` construct (line 2) by forcing the program to start the `while` loop (i.e., the next fragment of the code) only when 20 ms have passed since the invocation of function `foo`. Since the role of the first `stp` is to apply the offset, it does not enforce a deadline on the prior code and hence its second argument is set to $\infty$. The second `stp` (line 5) ensures that the `while` loop is periodic and is activated only when 60 ms have passed since the execution of the previous timing point. For example, in the timing diagram shown in Fig. 2, the first call to `work()` takes 30 ms and the program reaches the `stp` on line 5 at $t = 50$ ms. The `stp` on line 5 will delay the execution of the program until $t = 80$ ms to ensure that the `while` loop (line 3) is executed periodically every 60 ms. Furthermore, the `stp` on line 5 specifies a deadline for the code in the `while` loop. Since the deadline is relative to the previous time point, the first deadline will be at time $60 = 20 + 40$, the second deadline will be at $t = 120$ ms, etc. As an STP applies a soft deadline, Timed C will not stop the code if it overruns the deadline. For example, in Fig. 2, the second call to the `work()` function takes 45 ms (released at $t = 85$, finished at $t = 130$). The `stp` function will then return the amount of overrun (tardiness) experienced by the code fragments. It is worth noting that our toolchain ensures that, even if the code within the timing points overruns, it will not impact the arrival time of later timing points.

Firm timing points (FTPs) are defined as follows

```
ftp(expr1, expr2, n)
```

where the arguments `expr1`, `expr2` and `n` have the same meaning as for `stp`. However, unlike an STP, an FTP ensures that the execution of the code fragment prior to the timing point will be terminated at the deadline specified by `expr2`.

```
1 task work(){
2   while(1){
3     foo();
4     ftp(40,20,ms);
5     bar();
6     stp(20,10,ms);
7   }
8 }
```

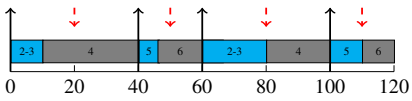



Fig. 3: Example of mixed soft and firm timing points. The example shows a generalized multiframe task implemented in Timed C.

In cases where the lower and upper bounds of a timing point are equal, Timed C provides the following constructs

```
sdelay(expr, n)        fdelay(expr, n)
```

which are equivalent to `stp(expr, expr, n)` and `ftp(expr, expr, n)`, respectively.

Fig. 3 shows a Timed C program that uses a mixture of soft and firm timing points and a timing diagram depicting one of its executions. In this timing diagram, the first call to the function `foo()` completes at $t = 10$. Since the `expr1` of the next timing point (line 4) is 40, Timed C delays the call to function `bar()` until time $t = 40$. The function `bar()` completes at time $t = 45$ and the `stp` on line 6 delays the next iteration of the `while` loop until $t = 60$ (= 40 + 20). In the second iteration, function `foo()` would have required 30 ms; however, thanks to the firm timing point (line 4), it is interrupted and aborted at $t = 80$, preventing the overrun from impacting subsequent invocations of the `bar()` function. Since in some scenarios interrupting a computation may lead to unexpected behavior, Timed C's `critical` construct, discussed shortly in Section III, can be used to defer such untimely interruptions.

In this paper, we focus on a subset of Timed C's expressive power to encode *generalized multiframe tasks* (GMFs) [5], where frames have either soft or firm deadlines. To this end, we next introduce a model of Timed C programs as GMF tasks.

### B. System Model

We consider a uniprocessor platform that runs a set of generalized multiframe tasks (GMF) [5] scheduled by a job-level fixed-priority (JLFP) scheduling algorithm such as fixed-priority (FP) or earliest-deadline first (EDF) scheduling. We assume that when a code fragment is called, it executes non-preemptively until completion or until it is terminated by a firm timing point. Each timing point also provides a preemption point in the application, i.e., when the timing point is reached another task may be dispatched by the scheduler.

Following the GMF task model [5], we assume that each task $\tau_i$ $(1 \leq i \leq n)$ is defined by an offset $O_i$ and consists of a finite sequence of $N_i$ frames $\langle \mathcal{F}_{i,1}, \mathcal{F}_{i,2}, \ldots, \mathcal{F}_{i,N_i} \rangle$. Each frame $\mathcal{F}_{i,j}$ is characterized by $(C^{min}_{i,j}, C^{max}_{i,j}, A_{i,j}, D_{i,j}, j_{i,j}, \gamma_{i,j})$, where $C^{min}_{i,j}$ is the best-case execution time (BCET), $C^{max}_{i,j}$ is the worst-case execution time (WCET), $A_{i,j}$ is the inter-arrival time, $D_{i,j}$ is the relative deadline, $j_{i,j}$ is the release jitter, and $\gamma_{i,j} \in \{\text{soft, firm}\}$ is the type of deadline of the frame. The first instance of the first frame $\mathcal{F}_{i,1}$ arrives at time $O_i$. Then, the value of $A_{i,j}$ denotes the relative time between the arrival of an instance of frame $\mathcal{F}_{i,j}$ and the arrival of the previous frame released by the same task, as explained

in Section II-A. To reduce clutter, we introduce the notation $O_{i,j} = O_i + \sum_{k=2}^{j} A_{i,k}$ to refer to the offset of frame $\mathcal{F}_{i,j}$. We also define the period of the GMF task as $T_i = \sum_{k=1}^{N_i} A_{i,k}$.

Each periodic GMF task generates an infinite number of instances. For the sake of analysis, we transform the set of periodic GMF tasks in an infinite set of jobs $\mathcal{J}$. Each job $J_i \in \mathcal{J}$ refers to the execution of one frame executed by a GMF task. To relate the two models, we define the three functions $\delta(J_i) : \mathbb{N} \to \{1, \ldots, n\}$, $\sigma(J_i) : \mathbb{N} \to \mathbb{N}$, and $\lambda(J_i) : \mathbb{N} \to \{1, \ldots, N_{\delta(J_i)}\}$ that return the task number, instance number, and frame number corresponding to job $J_i$, respectively.

Each job $J_i$ is characterized by a 7-tuple $(r_i^{min}, r_i^{max}, c_i^{min}, c_i^{max}, d_i, \gamma_i, \pi_i)$, where $r_i^{min}$ is the absolute arrival time (also known as the earliest release time), $r_i^{max}$ is the latest release time, $c_i^{min}$ and $c_i^{max}$ are the BCET and WCET, $d_i$ is the absolute deadline, $\gamma_i$ is the type of deadline, and $\pi_i$ is the priority of the job. Let $k$ denote the value returned by $\delta(J_i)$, $\ell$ denote the value returned by $\sigma(J_i)$, and $j$ denote the value of $\lambda(J_i)$. Then, $J_i$'s BCET and WCET are given by $c_i^{min} = C_{k,j}^{min}$ and $c_i^{max} = C_{k,j}^{max}$, respectively, and the earliest and latest release times are given by $r_i^{min} = O_{k,j} + T_k \cdot (\ell - 1)$, $r_i^{max} = r_i^{min} + j_{k,j}$. The deadline of $J_i$ is $d_i = r_i^{min} + D_{k,j}$. It is worth noting that the priority of a job depends on the scheduling policy. For example, for FP scheduling, $\pi_i$ is equal to the priority of the associated task $\tau_k$, while for EDF scheduling it is equal to the absolute deadline of the job, i.e., $\pi_i = d_i$, where a smaller value denotes a higher priority. We assume that ties on the priority are broken arbitrarily but consistently.

Each job has a non-deterministic yet bounded release time that occurs within the interval $[r_i^{min}, r_i^{max}]$. We assume that the release jitter is caused by implementation factors such as interrupt latency or timer inaccuracy.

We define $O^{max}$ to be the latest possible arrival time of the first instance of any frame in the task set, that is, $O^{max} = \max_i\{O_{i,N_i} \mid \forall i \in \{1, \ldots, n\}\}$. The hyperperiod of a set of GMF tasks is denoted by $H$ and is equal to the least-common multiple (LCM) of the periods of the tasks. We denote the task set utilization by $U = \sum C_i/T_i$, where $C_i$ is the total sum of the WCET of the frames of a task, i.e., $C_i = \sum_{1 \le j \le N_i} C_{i,j}^{max}$.

In the context of Timed C, a frame of a GMF task is a code segment between two timing points. Its inter-arrival time is defined by the argument `expr1` of the preceding timing point, while its relative deadline is defined by the argument `expr2` of the following timing point. An example of a GMF task with two frames implemented in Timed C is shown in Fig. 3.

## III. Timing Analysis

Our end-to-end toolchain currently uses a measurement-based timing analysis to simplify portability between different platforms. However, the overall end-to-end methodology itself is not bound to measurement-based methods, and static timing analysis would be interesting to add in future work.

In this section, we explain how the toolchain currently estimates the BCET, WCET, maximum jitter, and trigger precision for a given Timed C program and execution platform. We explain the meaning of these terms and how the values are computed. As depicted in the first part of Fig. 1, the timing analysis consists of two main stages: (i) *instrumentation* and (ii) *generation of timing traces*. The timing traces are then used to generate the input to the schedulability test.

### A. Instrumentation

There are two steps: **(i)** a Timed C instrumentation step that takes Timed C code as input and produces an instrumented version of the Timed C code, including measurement statements, and **(ii)** the source-to-source compilation that compiles the instrumented Timed C code to target-specific C code.

We explain the instrumentation procedure using the Timed C example program in Fig. 4(a). Task `rts` periodically calls functions `sense`, `compute`, and `actuate`. The function `sense` writes a new value to `s` by reading from a sensor. The function `compute` reads from `s` and `a`, and writes to `b`. In line 6, the content pointed to by `b` is copied to `a`. In this example, the calls to `sense` and `compute` have a firm deadline of 40 ms as specified by the `fdelay` at line 8. Note, when `compute` returns a new value and writes into `b`, the `critical` block ensures that this new value is written to `a` without being interrupted by `fdelay`. That is, the Timed C construct `critical` makes sure that the execution of the critical block has finished before the execution of the code fragment is aborted. The term *trigger precision* is used for the extra time it takes to escape out of the critical section and jump to the `fdelay`. On line 9, `actuate` reads `a` and performs its operation with a soft deadline of 10 ms. If `compute` and `sense` take longer than 40 ms, the previous value of `a` will be used by `actuate`. The periodicity is still correct.

The instrumenting compiler assigns a unique *id* to each timing points in a task. In the example, there are 3 timing points (marked as `TP0`, `TP1`, and `TP2` in the code), where the initial timing point `TP0` is the start of the function. There are three code fragments, each represented as the code between two timing points (fragments `TP0-TP1`, `TP1-TP2`, and `TP2-TP1`).

During the instrumentation, the compiler performs static analysis and inserts three different types of instructions to each code fragment. These instructions measure the absolute arrival, start time, and finish time of the code fragment. The instruction for measuring the arrival and start time are inserted at the beginning of the code fragment (for instance before line 2 for fragment `TP0-TP1`). The instruction for measuring the finish time is inserted at the end of the code fragment (before line 8 in the case of `TP0-TP1`).

The instrumenting compiler passes the instrumented Timed C code to the KTC compiler [31] that compiles the instrumented Timed C code into target specific code.

### B. Generation of Timing Traces

Timing traces are generated when the instrumented target-specific binary is executed. The traces can either be generated as a continuous log while running the real-time system on the target platform, or as a summary log that only exports the worst-case values, and not the whole trace.

```
1  task rts(int* a,int* b,int s){    // TP0
2    while(1){
3      sense(&s);
4      compute(b, a, s);
5      critical{
6        memcpy(a, b, 100);
7      }
8      fdelay(40, ms);               // TP1
9      actuate(a);
10     sdelay(10, ms);               // TP2
11   }
12 }
```

(a)

(b)

| *src* | *arrival* | *start* | *finish* | *precision* | *dst* |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 30 | 0 | 1 |
| 1 | 40 | 42 | 52 | 0 | 2 |
| 2 | 50 | 52 | 91 | 23 | 1 |
| 1 | 90 | 91 | 100 | 0 | 2 |

Fig. 4: **(a)** Example Timed C task and **(b)** a partial timing trace.

Fig. 4(b) shows an example of the former approach of a partial timing trace. Each row in the timing trace lists the execution details of a code fragment. For efficiency of the instrumented C code, this information is stored in a memory buffer before it is written to the timing trace log. Each row in the buffer contains six fields, corresponding to the columns in Fig. 4(b). The instrumentation for measuring the arrival time stores the *src* and *arrival* information, where *src* is the ID of the source timing point, and *arrival* is the arrival time of the timing point at the start of a code fragment. The instrumentation for measuring the start and finish time store the *start* and *finish* information, respectively. Both these calls log the total time elapsed since the start of the task (absolute time). The successor timing point stores the *precision* and *dst* information, where *precision* is the maximal time it can take to abort a job and *dst* is the identifier of the destination timing point. The precision field stores the *trigger precision*, which includes platform jitter and worst-case execution time of a critical section, if available.

From the timing trace, the BCET, WCET, and trigger precisions for each code fragment can be computed. Recall from Section II-B that a code fragment between two timing points in Timed C corresponds to a GMF frame. For instance, assume that we compute the values for the frame that corresponds to the code fragment between timing points 1 and 2 (`TP1` and `TP2`) in Fig. 4. The WCET is then $C_{1,k}^{max} = max(52-42, 100-91) = 10$, the BCET $C_{1,k}^{min} = min(52-42, 100-91) = 9$. The release jitter cannot be computed from the timing trace, and is instead considered a platform specific parameter, which can be obtained in platform-specific ways. For instance, on a Linux system, the `cyclictest` tool may be used.

## IV. Schedulability Analysis

The proposed end-to-end toolchain integrates a customized and extended version of the open-source[2] schedulability analysis for non-preemptive job sets proposed by Nasri and Brandenburg [30], which we denote as *NPA* for brevity. The reason why we chose NPA as the basis for our toolchain is that it is the only applicable response-time analysis to yield *per-job* response-time bounds, which is essential for enabling a weakly hard sensitivity analysis (as discussed later in Section V).

In the following, we summarize NPA, point out where it is insufficient for our purposes, and then describe our extensions. Due to space constraints, we assume the reader to have some familiarity with NPA at a high level and refer to [30] for details.

### A. Background and Overview

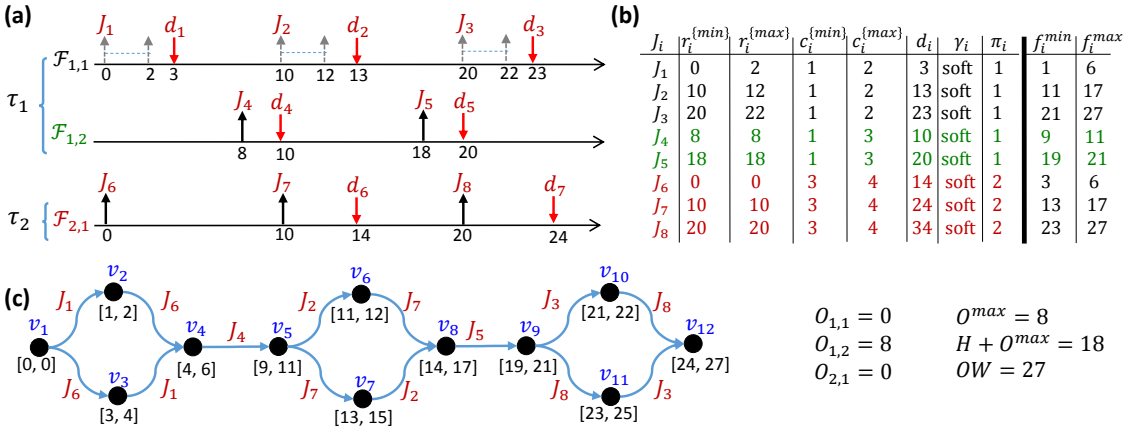
NPA [30] assesses the schedulability of a given *finite* set of jobs under a given JLFP scheduler by exploring the space of all possible schedules of the jobs that the scheduler can generate. NPA can be applied to analyze recurrent workloads (i.e., infinite job sets generated by, for instance, periodic tasks) if it is possible to determine a finite *observation window* such that the absence of deadline misses in the observation window implies the absence of deadline misses altogether.

To effectively search the extremely large space of possible schedules, NPA relies on the notion of a *schedule-abstraction graph*, which allows for an aggregation of schedules with similar properties (i.e., scenarios that lead to similar scheduling decisions). A schedule-abstraction graph is a directed acyclic graph (DAG) whose edges represent jobs (i.e., code segments executed non-preemptively) dispatched by the scheduling algorithm, and whose vertices represent time intervals during which the last-scheduled job on any incident path may complete its execution in any of the schedules represented by the path.

For example, Fig. 5 shows a job set and its equivalent schedule-abstraction graph under FP scheduling. This job set has been generated for an observation window of length 27 for two GMF tasks $\tau_1$ and $\tau_2$, where $\tau_1$ has higher priority than $\tau_2$. Each edge represents a job that is scheduled after a system state represented by its source vertex. Each path represents an ordering of the execution of different jobs. For example, the path $\langle v_1, v_2, v_4, v_5 \rangle$ represents the execution of $J_1$, $J_6$, and $J_4$ in that specific order. Each vertex $v_i$ is associated with an interval delimited by the earliest and latest finish time of the last job scheduled on any path that connects $v_1$ to $v_i$. For example, the earliest and latest finish time of $J_1$ when it is scheduled after $v_3$ are 4 and 6, respectively. Similarly, the earliest and latest finish time of $J_6$ when it is scheduled after $v_2$ are also 4 and 6, respectively. The fact that scheduling $J_1$ after $v_3$ and scheduling $J_6$ after $v_2$ lead to the same state (i.e., vertex) is essential for searching the space of all possible schedules (i.e., it is the result of deliberate search-space pruning).

The graph is explored using a breadth-first approach that alternates between two phases: *expansion* and *merging*. During the expansion phase, every leaf vertex is expanded by adding an edge for each of the not-yet-scheduled jobs that can potentially be scheduled after that leaf vertex by the given JLFP scheduling policy. For instance, in the example in Fig. 5, at time 0 (represented by the initial vertex $v_1$), jobs $J_1$ and $J_6$ are the only jobs that can potentially be executed after $v_1$. On the one hand, if $J_1$ is released at time 0, it will be the highest-priority ready job and hence will be dispatched, which is represented with a new vertex $v_2$ that is connected to $v_1$ with an edge labeled $J_1$. On the other hand, if $J_1$ is not released at time

[2]https://github.com/brandenburg/np-schedulability-analysis




| $J_i$ | $r_i^{\{min\}}$ | $r_i^{\{max\}}$ | $c_i^{\{min\}}$ | $c_i^{\{max\}}$ | $d_i$ | $\gamma_i$ | $\pi_i$ | $f_i^{min}$ | $f_i^{max}$ |
|---|---|---|---|---|---|---|---|---|---|
| $J_1$ | 0 | 2 | 1 | 2 | 3 | soft | 1 | 1 | 6 |
| $J_2$ | 10 | 12 | 1 | 2 | 13 | soft | 1 | 11 | 17 |
| $J_3$ | 20 | 22 | 1 | 2 | 23 | soft | 1 | 21 | 27 |
| $J_4$ | 8 | 8 | 1 | 3 | 10 | soft | 1 | 9 | 11 |
| $J_5$ | 18 | 18 | 1 | 3 | 20 | soft | 1 | 19 | 21 |
| $J_6$ | 0 | 0 | 3 | 4 | 14 | soft | 2 | 3 | 6 |
| $J_7$ | 10 | 10 | 3 | 4 | 24 | soft | 2 | 13 | 17 |
| $J_8$ | 20 | 20 | 3 | 4 | 34 | soft | 2 | 23 | 27 |

Fig. 5: A job set and its equivalent schedule-abstraction graph in the observation window $[0, 27]$.

0 due to experiencing release jitter (note that $r_1^{min} = 0$ and $r_1^{max} = 2$), then job $J_6$ will be the highest-priority ready job at time 0. This is recorded by the creation of $v_3$ with a finish-time interval $[3, 4]$ connected to $v_1$ with an edge labeled $J_6$. To determine the eligible jobs that can follow a given vertex during the expansion phase, NPA implements a set of rules that will be described with more detail in Section IV-C.

During the merge phase (which directly follows the expansion phase), all paths from the initial vertex to leaf vertices of the expanded graph that include the same set of jobs and have intersecting time intervals are merged. For example, in Fig. 5(c), after the expansion of $v_2$, a vertex $v_2' = [4, 6]$ connected by an edge with label $J_6$ will be created. Similarly, after the expansion of $v_3$, a vertex $v_3' = [4, 6]$ connected by an edge with label $J_1$ will be created. These two new leaf vertices are collapsed into a single vertex $v_4$ during the merge phase since the path from $v_1$ to $v_2'$ references the same set of jobs (i.e., $\{J_1, J_6\}$) as the path from $v_1$ to $v_3'$, and the intervals of $v_2'$ and $v_3'$ intersect. Thanks to the merge phase, at the end of each busy window, i.e., an interval of time where the processor may remain continuously busy, the graph has only one leaf vertex.

To summarize, for a given job set, NPA [30] iteratively explores its schedule-abstraction graph, which represents the space of all possible schedule. Each job's earliest- and latest-possible finish times can be trivially inferred from the graph.

### B. Challenges and Open Problems

Extending NPA to Timed C programs presents several challenges. First, NPA assumes that all jobs are independent; it is thus oblivious to the precedence constraints among frames of the same task. The issue appears if, for example, a task has a relative deadline exceeding its period (i.e., an arbitrary deadline), or when a job with a soft timing point becomes tardy or does not even start its execution by its deadline. If such jobs are falsely assumed to be independent, then the analysis may choose any of the not-yet-scheduled code segments of a task to execute next, including code segments whose predecessors have not yet been completed. As such, the analysis may include an *impossible* scenario in the schedule abstraction graph, which may lead to optimistic and hence possibly unsafe results.

Second, NPA assumes that all jobs run to completion, i.e., it does not support *segment abortion* at the boundary of `fdelay` timing points in Timed C programs. Clearly, aborting segments has a large impact on the space of possible schedules.

Third, to the best of our knowledge, prior work has not yet established a safe finite observation window $OW$ for soft real-time GMF tasks, without which NPA cannot be applied.

### C. Analysis Extensions

We next explain how we extended NPA to include code abortion and precedence constraints. In Section IV-D, we show how to obtain a safe observation window for GMF tasks.

*1) Supporting Code Abortion:*

*a) Job eligibility:* For each state reachable in the schedule-abstraction graph and for every incomplete job $J_i$, NPA computes the earliest- and latest-possible start times $EST_i$ and $LST_i$, respectively, to determine whether it may be dispatched next. As in the original analysis [30], a *soft-deadline* job (i.e., a job corresponding to a soft timing point) may be the next job scheduled if and only if:

$$EST_i \leq LST_i. \tag{1}$$

Ineq. (1) does not consider the job's deadline when checking whether it is eligible for execution. However, a *firm-deadline job* (i.e., a job corresponding to a firm timing point) never commences execution past its deadline. Therefore, a firm-deadline job $J_i$ may be the next job dispatched on the platform by the scheduler if and only if Eq. (1) is respected *and* its deadline is later than its earliest start time, i.e., $EST_i < d_i$. If the latter condition is not respected, then the job cannot be scheduled since it is not able to start before its deadline.

For example, suppose $J_1$ has a firm deadline in the example in Fig. 5. As a result, during the expansion phase of vertex $v_3$, $J_1$ will no longer be a candidate to follow $v_3$ since $EST_1 = 3$, which is not strictly less than its deadline $d_1 = 3$.

*b) Job completion time:* If $J_i$ is the next job scheduled on the platform, and no job abortion must take place (i.e., if $J_i$ has a soft deadline), the earliest and latest finish times $EFT_i$ and $LFT_i$, respectively, are simply given by $EFT_i = EST_i + c_i^{min}$ and $LFT_i = LST_i + c_i^{max}$.

In contrast, if $J_i$ has a firm deadline, it will be aborted by the scheduler when reaching its deadline. Hence, we must first compute the earliest and latest possible times $EAT_i$ and $LAT_i$ at which the job may be aborted because of overshooting its deadline. Those are respectively computed as $EAT_i = d_i + cl^{min}$ and $LAT_i = d_i + tr_i^{max} + cl^{max}$, where $cl^{min}$ and $cl^{max}$ are the minimum and maximum execution times of the cleanup and rollback functions called when a job is aborted, and $tr_i^{max}$ is the maximum delay that may be suffered by the interrupt triggering the abortion due to, for instance, the execution of a critical section or timer imprecision.

Consequently, the earliest and latest finish times of a firm-deadline job $J_i$ are given by the following equations:

$$EFT_i = \min\left\{EAT_i,\ EST_i + c_i^{min}\right\} \tag{2}$$

$$LFT_i = \min\left\{LAT_i,\ LST_i + c_i^{max}\right\} \tag{3}$$

For example, let $J_4$ have a firm deadline in the example of Fig. 5 and assume that both $cl^{max}$ and $tr_4^{max}$ are zero. During the expansion of vertex $v_5$, we will thus have $EST_4 = 8$, $LST_4 = 8$, $EAT_4 = 10$ and $LAT_4 = 10$. Hence, $EFT_4 = 9$ and $LFT_4 = \min\{10, 8+3\} = 10$, meaning that $J_4$ will complete at the latest by time 10.

The proof of correctness in [30] transfers to our extensions as they rely only on the fact that the earliest and latest start and finish times of the scheduled jobs are indeed lower and upper bounds on their actual release and completion times.

*2) Supporting precedence constraints:* Since we must support precedence constraints to analyze the timing behavior of Timed C programs, we denote the set of predecessors of a job $J_i$ by $pred(J_i)$. All jobs in $pred(J_i)$ must complete their execution before $J_i$ can start executing.

For *single-core* platforms, extending the analysis to support precedence constraints is rather straightforward. One must simply add the following rule to the eligibility conditions defined in the previous section: *a job $J_i$ is eligible for execution in system state $S$ only if all jobs in $pred(J_i)$ already appear in the path leading to state $S$ in the schedule abstraction graph.*

Since a job appears on a path in the graph if and only if it has been scheduled in the execution scenario modeled by that path, the condition boils down to checking that all jobs in $pred(J_i)$ have already been scheduled.

### D. Obtaining a Finite Observation Interval

NPA analyzes the response time of every job that may be released in a finite observation window that is *representative* of the recurrent workload (i.e., infinite job set) in the sense that analyzing the observation window is sufficient to ensure temporal correctness in general. However, to the best of our knowledge, no safe bound on the observation window length for soft real-time task systems is known form prior work. Since Timed C programs may contain soft timing points and hence generate jobs with soft deadlines, we must provide a condition that indicates when the analysis may safely be stopped.

In this section, we make the following assumptions, which are met in the context of our toolchain: **(1)** the analysis is run with $c^{min} = 0$ for all jobs, **(2)** the hyperperiod $H$ of the task set is known and, without loss of generality, **(3)** at least one task releases a job at time 0.

It is well-known that the job *release pattern* of a set of periodic tasks that respects (2) and (3) is periodic with period $H$ from time $O^{max}$ onward. Two cases must thus be considered:

1) *All jobs released in the hyperperiod finish their execution by time $H + O^{max}$.* In that case, there is no workload carried over from one hyperperiod to the next. Thus, the worst-case response time of every job in subsequent hyperperiods (i.e., every interval $[O^{max} + k \cdot H, O^{max} + (k+1) \cdot H)$ with $k \in \mathbb{N}$) will be lower than or equal to that in the interval $[O^{max}, H + O^{max})$. Consequently, the analysis can safely stop after the first hyperperiod extended by $O^{max}$.
2) *Some jobs released in the hyperperiod do not finish their execution by $H + O^{max}$.* In that case, there is some workload that is carried over from one hyperperiod to the next. Hence the analysis must continue since the system state at the beginning of the second hyperperiod is different from the system state at the beginning of the first hyperperiod. Now, let $t_{idle} > H + O^{max}$ be the first instant after $H + O^{max}$ such that the processor is certainly idle. That is, there is only one system state in the schedule abstraction graph with no job running at time $t_{idle}$ and all not-yet-scheduled jobs have an earliest start time $EST_i$ larger than $t_{idle}$. Then, the analysis can be safely stopped at time $t_{idle}$.

Case 2 in the above discussion relies on Assumption (3), that is, that the analysis is performed with $c^{min} = 0$ for all jobs until time $t_{idle}$. Since all jobs before $t_{idle}$ may execute for zero time, the case where the processor is idle is also one of the possible system states considered in the schedule abstraction graph for time $(t_{idle} - H)$. Therefore, all system states that may be reached from the state at $t_{idle}$ can also be reached from the system states already explored at time $(t_{idle} - H)$. The analysis can thus safely stop at $t_{idle}$.

Next, we derive an upper bound on time $t_{idle}$ by extending the classic notion of busy windows to our model. Specifically, we bound $t_{idle}$ with the following fixed-point recurrence:

$$t^{(0)} = 0 \tag{4}$$

$$t^{(1)} = \min_{\forall J_j \in J}\{r_j^{max}\} \tag{5}$$

$$t^{(i)} = t^{(i-1)} + \sum_{\{J_j \mid t^{(i-2)} < r_j^{max} \leq t^{(i-1)}\}} c_j^{max} \tag{6}$$

The recursion stops if the following condition is satisfied.

$$(t^{(i)} = t^{(i-1)}) \ \wedge\ (t^{(i)} \geq H + O^{max}) \wedge (\nexists J_j, r_j^{min} \leq t^{(i)} < r_j^{max}) \tag{7}$$

If at any point, $t^{(i)} = t^{(i-1)}$ but $\exists J_j, r_j^{min} \leq t^{(i)} < r_j^{max}$ or $t^{(i)} < H + O^{max}$, then the next $t^{(i)}$ is advanced to the beginning of a busy window as follows.

$$t^{(i)} = \min\{r_j^{max} \mid t^{(i)} < r_j^{max}\} \tag{8}$$

**Algorithm 1:** Bounding the *observation window* length

```
1  t_p ← 0;
2  t ← min{r_j^max};
3  while t ≠ t_p do
4  |   t_n ← t + Σ{c_j^max | t_p < r_j^max ≤ t};
5  |   if (t_n = t) ∧ (∃J_j | r_j^min ≤ t_n < r_j^max ∨ t_n < H + O^max)
   |     then
6  |   |   t_n ← min{r_j^max | t_n < r_j^max};
7  |   end
8  |   t_p ← t;
9  |   t ← t_n;
10 |   if t > t_threshold then
11 |   |   Abort;
12 |   end
13 |   return t;
14 end
```

Eq. (8) guarantees that the search for the end of a busy window continues until $t^{(i)}$ becomes larger than or equal to $H + O^{max}$. Further, if $t^{(i)} = t^{(i-1)} \wedge \nexists J_j, r_j^{min} \leq t^{(i)} < r_j^{max}$, then the processor is certainly idle at $t^{(i)}$ because $t^{(i)} = t^{(i-1)}$ entails that any job that has been certainly released before $t^{(i)}$ has certainly completed by $t^{(i)}$, and $\nexists J_j, r_j^{min} \leq t^{(i)} < r_j^{max}$ entails that there is no job that *may be released* before or at time $t^{(i)}$. Hence, when the recursion ends, $t^{(i)}$ is equal to an instant where the processor is certainly idle in any possible execution scenario. It thus upper-bounds $t_{idle}$ and hence is a safe bound on the length of the observation window.

Algorithm 1 summarizes the computation of the fixed-point iteration to obtain an upper bound on $t_{idle}$ using the equations derived above. As such fixed point is not guaranteed to exist (e.g., if the processor is overloaded), Algorithm 1 is *aborted* if no fixed point is found before a configurable threshold is reached and no further analysis of the workload is possible. If Algorithm 1 terminates before reaching the threshold, then it returns a valid upper bound on the observation window length.

## V. Sensitivity Analysis

Our end-to-end toolchain analyzes the sensitivity of the system w.r.t. each frame's WCET under weakly hard timing constraints. A weakly hard real-time task $\tau_i$ is feasible if any $k_i$ consecutive jobs of a task exhibit at most $m_i$ deadline misses.[3] The toolchain calculates the *WCET margins* for the strongest still-satisfied $(M, \mathcal{K})$ specification, where $M = m_1 + \cdots + m_N$ and $\mathcal{K} = \{k_1, \ldots, k_N\}$. That is, given a window length $k_i$ for each task, it bounds the largest *scaling factor* $\Delta^{max}$ by which all WCETs can be scaled so that the sum of the individual deadline-miss bounds $m_1, \ldots, m_N$ does not exceed $M$, where each $m_i$ is determined w.r.t. the given $k_i$.

### A. Motivating Example

Consider a GMF task set $\{\tau_1, \tau_2\}$, where $\tau_1$ has two frames with arrival times $A_{1,1} = 10$ ms and $A_{1,2} = 30$ ms (and hence $T_1 = 40$), and $\tau_2$ has one frame with arrival time $A_{2,1} = 50$ ms (and hence $T_2 = 50$). Both tasks have an offset of zero. Thus, the observation window of this task set is $H = 200$ and includes five instances of $\tau_1$ (each with two frames) and four single-frame instances of $\tau_2$. Suppose we set $k_1 = k_2 = 2$ to gain insight into whether back-to-back misses are possible.

[3]Bernat et al. [18] defined an $(m, k)$ weakly hard real-time constraint to denote that a least $m$ jobs *meet* their deadlines in any $k$ consecutive executions. In contrast, we define $m$ to show the number of *missed* deadlines since it simplifies the presentation and reasoning about the analysis.

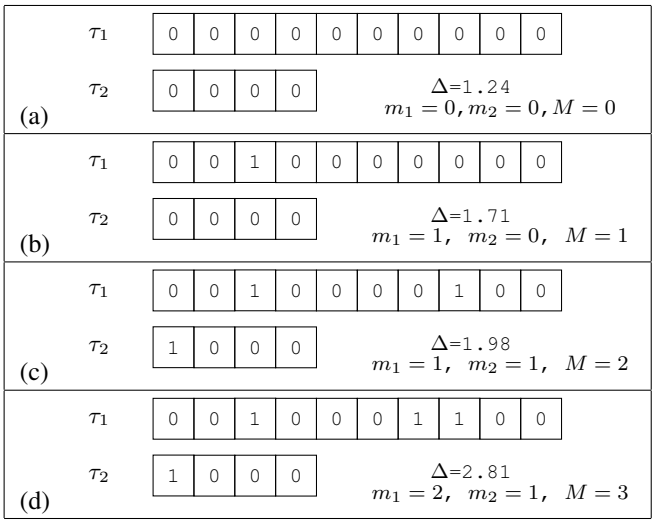



Fig. 6: The outcome of executing two tasks $\tau_1$ and $\tau_2$ as the WCETs of the two tasks are increased.. A value of 1 denotes a deadline miss.

Fig. 6(a) shows the analysis outcome for the observation window assuming WCETs are scaled up by $\Delta = 1.24$, where no deadline misses occur yet. Thus, $M = m_1 + m_2 = 0$.

After further increasing the WCETs by a factor of $\Delta = 1.71$, we observe a first deadline miss in $\tau_1$, as indicated in Fig. 6(b). We thus have $m_1 = 1$ and $m_2 = 0$, yielding $M = 1$.

Next, in the scenario shown in Fig. 6(c), $\Delta$ has been increased to 1.98. As a result, in addition to the 3rd job of $\tau_1$, its 8th job now also misses its deadline. However, despite the two deadline misses, we still have $m_1 = 1$ since $m_1$ is the largest number of deadline misses in any two *consecutive* jobs of $\tau_1$ (recall that $k_1 = 2$). The 1st job of $\tau_2$ now also misses its deadline, which implies $m_2 = 1$. Hence, $M = 1 + 1 = 2$.

Finally, Fig. 6(d) shows an execution scenario in which the 7th instance of $\tau_1$ also misses its deadline due to a further increase of all WCETs by a factor of $\Delta = 2.81$. Now, the largest number of deadline misses in any two consecutive jobs of $\tau_1$ is 2, and therefore $M = m_1 + m_2 = 2 + 1 = 3$.

The example highlights how more deadline misses become possible as WCETs increase. The overall goal of the $(M, \mathcal{K})$ sensitivity analysis is to report the largest WCET increase (i.e., the largest scaling factor $\Delta$) that does not cause $M$ to exceed a given threshold. The analysis also yields for each task the "steps" of its $m_i$ value, which allows developers to reason about the implications of WCET overruns on each individual task.

### B. Sensitivity Analysis Overview

Our toolchain performs an $(M, \mathcal{K})$ sensitivity analysis by computing scaling factors between 0 and an upper bound that results in a change to $M$ (for a given set $\mathcal{K}$). Fig. 7 illustrates how $M$ relates to the scaling factor $\Delta$. For example, around

**Algorithm 2:** `sensitivityAnalysis()`

$M_{sup} \leftarrow$ `calcMisses`$(\Delta_{sup})$;
$\Delta^{min} \leftarrow \{\Delta_{sup} |\ i \in 0..M_{sup}\}$;
$\Delta^{min}_0 \leftarrow 0$;
$\Delta^{max} \leftarrow \{0 |\ i \in 0..M_{sup}\}$;
$\Delta^{max}_{M_{sup}} \leftarrow \Delta_{sup}$;
$i \leftarrow 0$;
**while** $i < M_{sup}$ **do**
  $u \leftarrow$ `min`$(\{j |\ i < j < M_{sup} \wedge \Delta^{min}_j < \Delta_{sup}\} \cup \{M_{sup}\})$;
  $(\Delta^{min}, \Delta^{max}, i) \leftarrow$ `bsearch`$(\Delta^{min}, \Delta^{max}, i, u)$;
**end**

**Algorithm 3:** `bsearch`$(\Delta^{min}, \Delta^{max}, l, u)$

**if** $(\Delta^{min}_u - \Delta^{max}_l < \epsilon)$ **then**
  **return** $(\Delta^{min}, \Delta^{max}, u)$;
**end**
**else**
  $\Delta_{mid} \leftarrow \dfrac{\Delta^{min}_u + \Delta^{max}_l}{2}$;
  $M \leftarrow$ `calcMisses`$(\Delta_{mid})$;
  $\Delta'^{min} \leftarrow \Delta^{min}$;
  $\Delta'^{max} \leftarrow \Delta^{max}$;
  $\Delta'^{min}_M \leftarrow$ `min`$\big(\{\Delta_{mid}, \Delta^{min}_M\}\big)$;
  $\Delta'^{max}_M \leftarrow$ `max`$\big(\{\Delta_{mid}, \Delta^{max}_M\}\big)$;
  $u' \leftarrow \begin{cases} u, & l = M \\ M, & l \neq M \end{cases}$;
  `bsearch`$(\Delta'^{min}, \Delta'^{max}, l, u')$;
**end**

**Algorithm 4:** `calcMisses`$(\Delta)$

$\mathcal{J}' \leftarrow \mathcal{J}$;
$c'^{max} \leftarrow \{\Delta \cdot c^{max}_i |\ i \in 1..|\mathcal{J}'|\}$;
$(F'^{min}, F'^{max}) \leftarrow$ `schedulabilityAnalysis`$(\mathcal{J}')$;
$M \leftarrow 0$;
**foreach** $i \in 1..|\mathcal{K}|$ **do**
  $\mathcal{L} \leftarrow \{d'_j - f'^{max}_j |\ j \in 1..|\mathcal{J}'|, i = \delta(j)\}$;
  $M' \leftarrow 0$;
  **foreach** $j \in 1..|\mathcal{L}|$ **do**
    $M' \leftarrow \texttt{max}\left(M', \sum_{p=j}^{j+k_i-1} \begin{cases} 0, & L_{f(p)} \geq 0 \\ 1, & L_{f(p)} < 0 \end{cases}\right)$,
    where $f(x) =$
    $(x-1) \bmod |\mathcal{L}| + 1$ and $L_{f(x)} \in \mathcal{L}$;
  **end**
  $M \leftarrow M + M'$;
**end**
**return** $M$;

$\Delta \approx 1.5$, there is a discontinuity where $M$ increases from 0 to 1. Another "step" from 1 to 2 occurs around $\Delta \approx 1.8$.

To reason about these ranges, we let $[\Delta^{min}_x, \Delta^{max}_x]$ denote an interval such that $M = x$ if $\Delta \in [\Delta^{min}_x, \Delta^{max}_x]$. For example, $M = 1$ as long as $\Delta \in [\Delta^{min}_1, \Delta^{max}_1]$. Conversely, the step from 1 to 2 occurs for some $\Delta \in (\Delta^{max}_1, \Delta^{min}_2)$. The goal of our sensitivity analysis is to find a close lower bound on $\Delta^{max}_x$ for each $x \in \{0, 1, 2, \ldots\}$, up to some upper bound $\Delta_{sup}$ on the maximum scaling factor of interest. The $\Delta_{sup}$ value is computed by considering **(i)** a user-specified *utilization cap* on the acceptable total utilization of the system (100% by default), **(ii)** a user-provided *limit of interest* $l_i$ for each task $\tau_i$, which

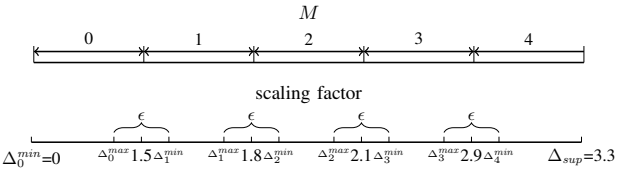



Fig. 7: The relation between $M$ and scaling factor $\Delta$

| | |
|---|---|
| The maximum initial upper bound for $\mathcal{J}$* | $\Delta_{sup}$ |
| The total number of deadline misses | $M$ |
| The mapping of misses to its minimum scaling factor | $\Delta^{min} = \{\Delta^{min}_0, \ldots, \Delta^{min}_M\}$ |
| The mapping of misses to its maximum scaling factor | $\Delta^{max} = \{\Delta^{max}_0, \ldots, \Delta^{max}_M\}$ |
| The number of jobs in observation window for $\tau$ | $\mathcal{L} = \{L_1, \ldots, L_N\}$ |
| The output of the schedulability analysis | $(F^{min}, F^{max})$ |
| The maximum permissible error* | $\epsilon$ |
| The number of consecutive window for $\tau$* | $\mathcal{K} = \{k_1, \ldots, k_N\}$ |
| Function mapping job id to its task id* | $\delta : \mathbb{N} \rightarrow \mathbb{N}$ |
| Set of jobs, in a given observation window* | $\mathcal{J}$ |

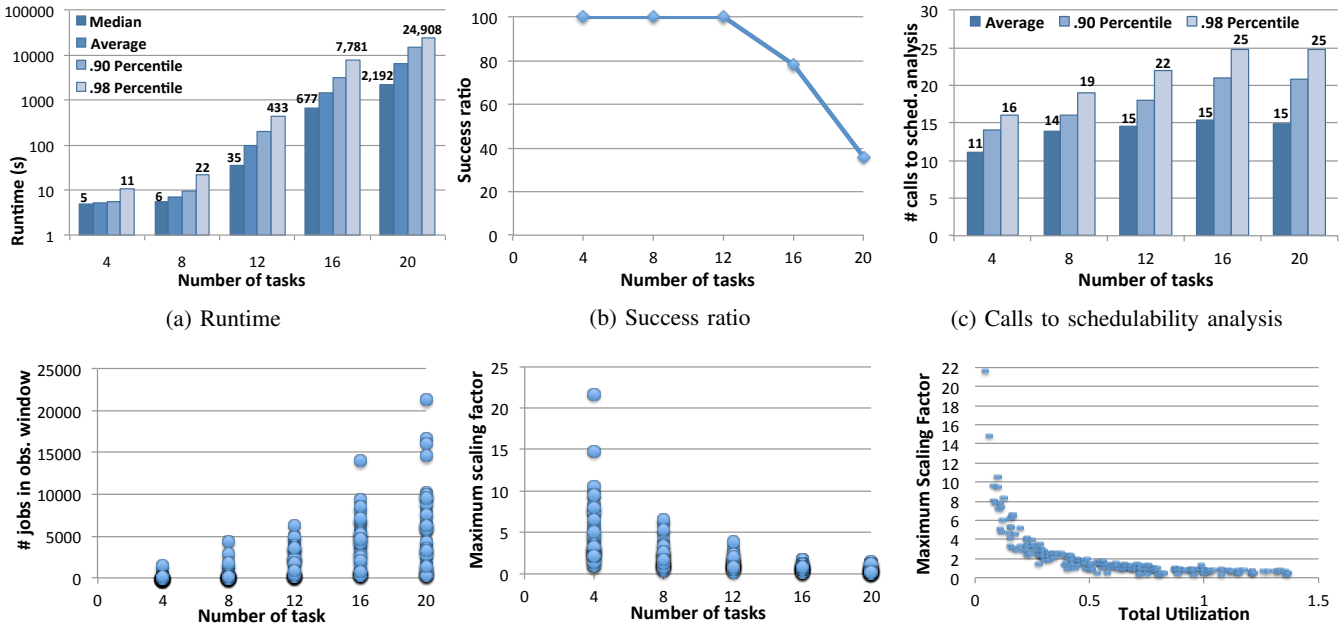
Fig. 8: Notation. The four last rows marked with (*) specify implicit values that are available in all algorithms.

bounds the largest $m_i$ value that is of interest to the user (by default, $l_i = k_i$), and **(iii)** by computing an upper limit based on the slack of the original task set. To speed up this process, random simulations of the observation window (rather than the analysis presented in Section IV) are used to *estimate* a lower bound on each $m_i$ (and hence $M$) since determining $\Delta_{sup}$ is not soundness-critical. The complete sensitivity analysis is given in Algorithms 2 to 4. Fig. 8 summarizes our notation.

We explain the analysis with the example in Fig. 7, in which $\Delta_{sup} = 3.3$. The sensitivity analysis computes the maximum scaling factor $\Delta^{max}_M$ for each $M \in \{0, 1, 2, 3\}$ as follows. Suppose $M = 1$ and $\Delta \in [1.5, , 1.8]$. For each $M$, the algorithm initializes $\Delta^{max}_M$ to 0 and $\Delta^{min}_M$ to $\Delta_{sup}$. Since we already know that the minimum scaling factor for $M = 0$ is 0 and that the maximal scaling factor for $M = 4$ is 3.3, the algorithm updates $\Delta^{min}_0$ to 0 and $\Delta^{max}_4$ to 3.3. In the next step, for each possible value of $M \in \{0, 1, 2, 3\}$, the sensitivity analysis determines $\Delta^{min}_M$ and $\Delta^{max}_M$ with a binary search.

To compute $\Delta^{max}_M$, the sensitivity analysis specifies the search boundaries between the current value of $\Delta^{max}_M$ and an upper bound that is less than or equal to $\Delta_{sup}$. Although the binary search executes between two scaling factors, the sensitivity analysis specifies the search boundaries in terms of $M$, which in turn is used to calculate the boundaries of the scaling factor, for reasons that will become apparent later in this section. For instance, to compute $\Delta^{max}_0$ the algorithm searches between $\Delta^{max}_0 = 0$ and $\Delta^{min}_4 = 3.3$. For the given search boundaries, the algorithm computes the scaling factor in the middle, denoted $\Delta_{mid}$. It computes $M$ from $\Delta_{mid}$ based on the schedulability analysis results (Section IV), assuming the WCET of the input is scaled by $\Delta_{mid}$.

Continuing the example, during its first iteration while computing $\Delta^{max}_0$, the binary search algorithm computes $\Delta_{mid} = (0 + 3.3)/2 = 1.65$. From the schedulability analysis, we obtain that a scaling factor 1.65 results in $M = 1$. Then both $\Delta^{max}_1$ and $\Delta^{min}_1$ are updated to 1.65. Since $M \neq 0$ at $\Delta_{mid}$, the second iteration executes with updated boundaries 0 and 1 between scaling factors $\Delta^{max}_0 = 0$ and $\Delta^{min}_1 = 1.65$. In the second iteration, $\Delta_{mid}$ is computed as $0.83 = (0 + 1.65)/2$.




(a) Runtime (b) Success ratio (c) Calls to schedulability analysis

(d) Problem size (e) Scaling factor vs. number of tasks (f) Scaling factor vs. utilization

Fig. 9: Results of the scalability experiment. Inset (f) shows the maximum scaling factor ($\Delta_{sup}$) relative to the total utilization of *unscaled* task sets. Task sets with a utilization larger than one are scaled down and the maximum scaling factor is thus below one.

Suppose that the schedulability analysis indicates $M = 0$ for $\Delta = 0.83$; then $\Delta_0^{max}$ is updated to $0.83$. Note that $\Delta_0^{min}$ is not updated since its current value $0$ is less than the computed value $0.83$. Since $M$ at $\Delta_{mid}$ is 0, the third iteration executes with search boundaries $0$ and $1$. However, since $\Delta_0^{max}$ was updated in this iteration, the next step is performed between scaling factors $\Delta_0^{max} = 0.83$ and $\Delta_1^{min} = 1.65$. In the third iteration, $\Delta_{mid}$ is computed as $1.24 = (0.83 + 1.65)/2$. If $M = 0$ for $\Delta = 1.24$, then $\Delta_0^{max}$ is updated to $1.24$. Finally, since $\Delta_1^{min} - \Delta_0^{max} = 1.65 - 1.24 = 0.41$ is less than $\epsilon = 0.5$, the algorithm terminates. As the largest observed scaling factor resulting in $M = 0$ is $\Delta_0^{max} = 1.24$, the end-to-end toolchain reports $24\%$ as the WCET margin for $M = 0$.

## VI. Evaluation

We evaluated two aspects. First, in VI-A, we report on an experiment conducted to evaluate the scalability of the end-to-end toolchain (w.r.t. the number of tasks). Thereafter, in VI-B, we present a case study that exhibits how the toolchain provides useful insights into an application's temporal robustness.

### A. Scalability

*Experimental Setup:* We ran the end-to-end toolchain within a Docker environment on an Intel Xeon Gold 6148 CPU clocked at 2.40 GHz. Timing traces were obtained by executing the instrumented Timed C code on a Raspberry Pi 2 Model B, which features an ARM Cortex A7 CPU clocked at 900 MHz, running the Raspbian Linux distribution with the PREEMPT_RT kernel patch in place. In all our experiments, we applied the Rate-Monotonic scheduling policy. Non-preemptive scheduling was enacted on Raspbian OS by assigning each code fragment a reserved (i.e., the highest) priority at the start of its execution, and reverting to its normal priority at the next preemption point. The system calls necessary for these priority adjustments are automatically inserted by the Timed C compiler.

Task sets were generated randomly following the period distribution of an automotive benchmark [25]. Recall that the period of a GMF task is the sum of the periods of its frames. The periods were sampled from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms according to the distribution reported by Kramer *et al.* [25]. The experiment was carried out for 4 to 20 tasks in increments of 4. In total, we generated 250 different valid task sets (50 each for $N \in \{4, \ldots, 20\}$). A generated task set was treated as invalid and discarded if $\Delta_{sup}$ is close to zero (less than epsilon) and some task's limit of interest is violated (i.e., $m_i > l_i$). The number of frames in each task ranged from 1 to 4. We randomly varied the workload by repeatedly running selected functions from MiBench [20].

Execution-time estimates were obtained by observing $100$ iterations of each task on the target platform. We used Linux's `cyclictest` tool to measure the platform-induced release jitter on the target platform and observed a maximum jitter of $356$ $\mu s$ in a 30-minute benchmark. In this evaluation, we used $0.98$ as the utilization cap (leaving $2\%$ of the system's capacity for OS overheads), $4$ as the limit of interest, $10$ as the window of interest ($k$), and $0.05$ as the epsilon resolution.

*Results:* Fig. 9a shows the runtime of the end-to-end toolchain as a function of the number of tasks. Fig. 9b shows the success rate of the end-to-end toolchain, where a run is considered successful when a call to the schedulability analysis completes without timeout (30 minutes). Figs. 9a and 9b show that task sets consisting of at most 12 tasks could be processed

by the end-to-end toolchain within a few seconds to a few minutes. They also show that, as the number of tasks increases, the runtime of the toolchain and number of timeouts increases considerably, too. Fig. 9c depicts the number of invocations of the schedulability analysis as a function of the number of tasks. Though there is only a modest increase in the number of calls of the schedulability analysis, the time required for each invocation grows substantially as the problem size increases (see Fig. 9d). Additionally, the maximum scaling factor computed by the sensitivity analysis pushes each task set to the limit of its schedulability, which in turn drastically increases the number of system states and possible schedules that the schedulability analysis must explore. As a result, timeouts become more frequent for larger workloads (Fig. 9b).

Figs. 9e and 9f show the maximum scaling factor as a function of the number of tasks and the total utilization, respectively. Unsurprisingly, the maximum scaling factor decreases as the system's capacity is approached by increasing either the number of tasks or per-task utilizations, which reflects that more heavily-loaded systems have less slack, and are thus temporally less robust (i.e., WCET margins are smaller). A key contribution of our end-to-end toolchain is to *quantify* this effect, and to expose it to the developer in a simple and actionable manner, as we highlight next with a case study.

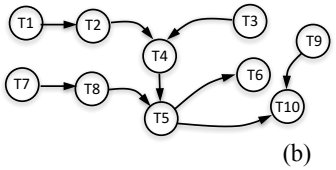
### B. Case Study

To demonstrate the benefits of the overall methodology, we constructed a synthetic case study inspired by an autonomous unmanned aerial vehicle (UAV). The parameters and dependencies of the task graph are based on the Paparazzi project [32]. This case study is synthetic since, although the task set and its parameters are based on a real project, the actual tasks that execute on our embedded platform are nonfunctional dummy tasks for ease of experimentation, and the experiment does not involve an actual UAV in operation. This simplifying approach is sufficient for our purposes since only the timing analysis and the task graph are needed to study how the proposed methodology and weakly hard sensitivity analysis can be used to explore the temporal robustness of a realistic task set.

*Task set:* The Paparazzi UAV is controlled by two 16 MHz microcontrollers [32]. In this case study, we study a consolidated setup where all tasks run on a single 32-bit 80 Mhz microcontroller, as shown in Figs. 10a and 10b. There are ten periodic tasks, with task T4 being a multi-frame tasks consisting of four frames. In comparison to the original Paparazzi task set, our workload features two additional tasks that estimate the UAV's position when GPS signals are not available. Each task is realized as a separate Timed C task, where task T4 models dependencies among its four subtasks (frames).

*Experimental Setup:* We obtained the timing traces by executing the instrumented Timed C implementation on a ChipKIT Max32 board with a 80 MHz 32-bit MIPS processor, running FreeRTOS version 10.2.1, with non-preemptive cooperative FP scheduling. The sensitivity analysis was performed on a MacBook Pro with two 3.1 GHz cores and 16 GB memory, running Docker Desktop Community 2.1.0.3. In our case study,

(a)




(b)

| ID | Description | Frames | Frequency |
|---|---|---|---|
| T1 | Distance and LIDAR sensors | 1 | 4Hz |
| T2 | Particle filter | 1 | 4Hz |
| T3 | Receiving GPS signals | 1 | 4Hz |
| T4 | Control (set of subtasks) | 4 | 4Hz |
| T5 | Stabilization | 1 | 20Hz |
| T6 | Reporting task | 1 | 10Hz |
| T7 | Receive radio commands | 1 | 40Hz |
| T8 | Manage radio commands | 1 | 40Hz |
| T9 | Check for fail-safe handling | 1 | 20Hz |
| T10 | Transmit to servos | 1 | 20Hz |

(c)

| Misses | WCET Margin |
|---|---|
| 0 | 1.55456852792 |
| 1 | 1.8654822335 |
| 2 | 2.02093908629 |
| 3 | 2.17639593909 |
| 5 | 2.25412436548 |
| 6 | 2.33185279188 |
| 7 | 2.40958121827 |

Fig. 10: Task set used in the UAV case study, based on the Paparazzi UAV project [32]. Edges indicate dependencies among tasks.

the (synthetic) tasks exhibited a total utilization of less than one on the deployment platform. That is, intuitively, the system has *some* slack to cope with WCET increases—but just how much of a margin for error is there? Our toolchain answers exactly this question in a developer-friendly, easy-to-understand way.

*Results and discussion:* Three aspects are of interest: **(i)** how much of a WCET margin is there before any of the critical tasks miss a deadline, **(ii)** what are the margins of the tasks that are allowed to experience a limited number of misses, and **(iii)** how would the margins change if certain tasks are optimized and their execution time reduced? To express that most tasks should not miss their deadlines, we configured the tool with $k_i = 1$ and the limit of interest $l_i = 1$ for all tasks except for tasks `T2` and `T6`. To ensure that task `T2` (particle filter) does not miss more than 3 deadlines in a row, we set $k_2 = 3$ and $l_2 = 3$. Configured like this, and with a utilization cap of $0.98$, the toolchain requires less than 7 s of runtime to report the WCET margins listed in Fig. 10c.

We observe that all WCETs can be scaled by factor of up to 1.5545 (55.45%) without resulting in any misses, which gives the developer an intuitively meaningful, *quantified* notion of the system's temporal robustness due to the available slack. At most one miss ($M = 1$) can occur until the WCET increase exceeds a factor of $1.8654$. By inspecting the per-task tool output (omitted from Fig. 10c for space reasons), we see that $m_1 = 1$, which means that the temporal correctness of (only) task `T1` can no longer be guaranteed at this point. Similarly, we can inspect what tasks are the reasons for different margins. For instance, the margin until the particle filter incurs up to $m_2 = 2$ misses within any window of $k_2 = 3$ jobs is $2.2541$.

In summary, this brief case study illustrates how an engineer can use the information provided by the toolchain to optimize the temporal robustness of a system and gain detailed insights into *which tasks* are vulnerable to experience deadline misses, in *which order* tasks will experience a degradation in temporal robustness, *how frequent* deadline misses become as WCETs

increase, and, perhaps most importantly, *how large* the margin of error w.r.t. WCET estimates is before anything goes wrong. Inarguably, such details are more useful to engineers than a simple "yes/no" analysis result based on uncertain assumptions.

## VII. Related Work

*Programming with time:* Several programming languages (such as Esterel [7], Lustre [33], Signal [26], Ada [12], and the real-time extension to Java, RTSJ [29]) and modeling languages (such as UML MARTE [1], Modelica [2], and Ptolemy [34]) provide explicit timing constructs. However, most small-scale embedded systems are programmed in C, which lacks timed semantics. Adding a notion of time to C programs can be done by either using APIs provided by operating systems [11] or language extensions such as Real-Time Concurrent C [17] and Timed C [31]. Of these, Timed C is an attractive choice for our purpose since with only a few timing constructs it is general enough to support various task models such as GMF tasks.

*Timing Analysis:* There is a rich established literature on WCET analysis of real-time applications, and many static timing analysis tools are available (e.g., [4, 6, 10, 15, 24, 28]. However, a major challenge for any of these tools is the analysis of modern complex hardware architectures. As a consequence, recent research has also focused on timing-predictable hardware [41, 48]. In contrast to the state of the art in timing analysis, the pragmatic measurement-based approach adopted in this paper is rather simplistic. Timing analysis in itself is not a contribution of this paper, but the integration of a timing analysis component is essential to obtaining an *end-to-end* toolchain. In future work, it will be interesting to also incorporate support for static WCET analysis tools as an alternative to the current measurement-driven approach.

*Schedulability Analysis:* Despite much prior work on the schedulability analysis of non-preemptive tasks on uniprocessor platforms [8, 13, 14, 23, 30, 39, 40, 42, 46], only few provide exact results. Further, none of these analyses yield *per-job response-time bounds* as required for our weakly hard sensitivity analysis (i.e., they yield task-level bounds, which are insufficient for $(m, k)$ compliance). The only prior exact analysis that provides per-job response times [30] supports neither precedence constraints nor job abortions (due to firm timing points). We provide the needed extensions in Section IV.

*Sensitivity Analysis:* Sensitivity analysis of timing parameters of real-time tasks, such as deadline, period, WCET, and offsets, has been studied by many researchers [3, 9, 21, 22, 27, 35–37, 43, 47]. These methods, however, are limited to the preemptive execution model and do not support weakly hard timing constraints, job abortion, or multiframe tasks as they occur in Timed C programs. For systems with limited preemptions, Regehr [38] proposed a random testing approach to analyze the sensitivity of schedulability w.r.t. a tasks' priority, preemption threshold, and WCET. However, Regehr's solution does not consider precedence constraints or firm deadlines. A system-level sensitivity analysis of WCETs w.r.t. end-to-end deadlines based on a binary-search approach was later proposed by Racu et al. [36, 37] and integrated in SymTA/S [21, 36].

## VIII. Limitations, Extensions, and Conclusion

We have introduced an end-to-end toolchain that integrates **(i)** a programming language with timed semantics, **(ii)** an analysis-aware compiler, **(iii)** measurement-based timing, **(iv)** schedulability, and **(v)** sensitivity analysis. Besides the integration aspects, the novel contributions of this paper are extensions w.r.t. (iv) and the first weakly hard solution for (v).

By design, any of these components can be replaced or improved individually. First off, our toolchain is based on the Timed C language and the KTC compiler, but the basic end-to-end principle is not limited to Timed C in any way. Another language or compiler may be trivially incorporated as long as it can generate sufficient instrumentation and metadata for timing and schedulability analysis; the engineering effort would be limited to adding support for outputting job sets in the simple file format used by the schedulability and sensitivity analyses.

A key feature of Timed C is that it is not a new programming language, but rather a small extension to C with a limited set of constructs for expressing timing constrains. Hence, a legacy program written in C without explicit timing constrains does not have to be translated into Timed C. Instead, the recommended approach is to view a Timed C program as a *coordination program* that defines the task set, including timing constrains such as periodicity and deadlines. Since the KTC source-to-source compiler needs to analyze the task set as a whole, it currently limits the scope of a Timed C program to one compilation unit. However, existing legacy C functions (without timing constructs) can simply be compiled separately and linked in, and then called from Timed C tasks.

The current toolchain relies on a measurement-based approach to timing analysis, but this is a purely pragmatic choice motivated by our target platforms and not a conceptual limitation of the toolchain. However, actually incorporating support for an off-the-shelf static timing analysis tool will likely require both significant engineering effort and further research. In particular, a static analysis of Timed C programs would need to **(i)** extract the code fragments between timing points in a format suitable for static timing analysis, and **(ii)** generate sound flow facts for the extracted code fragments.

The sensitivity analysis is currently the primary scalability bottleneck due to repeated invocations of the underlying schedulability analysis on difficult problem instances. While the schedulability analysis is conceptually an easy-to-replace component, there currently exists no viable alternative because the weakly hard sensitivity analysis requires *per-job* response-time bounds, for which there is little applicable prior work. In future work, it would be interesting and beneficial to devise a faster sensitivity analysis that does not rely on binary search or repeated invocations of the schedulability analysis.

Furthermore, the current schedulability analysis limits our toolchain to limited-preemptive scheduling and regular job-release patterns (e.g., GMF tasks). For the targeted small microcontroller platforms, this is an ideal choice, but for larger platforms (e.g., powerful multicore platforms) support for preemptive and/or sporadic tasks will need to be developed.

## IX. Acknowledgments

The authors would like to thank Daniel Lundén, Elias Castegren, Viktor Palmkvist, Oscar Eriksson, and Christian Schulte for their valuable input and feedback. We also thank the anonymous reviewers and the shepherd for their insightful comments and suggestions. This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032) as well as by national funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234), by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement through the European Regional Development Fund (ERDF), and by national funds through FCT within project POCI-01-0145-FEDER-029119 (PReFECT).

## References

[1] "MARTE," http://www.omg.org/spec/MARTE/, accessed: 2019-04-10.

[2] "Modelica and the Modelica Association Modelica Association," https://www.modelica.org, accessed: 2019-04-10.

[3] P. Balbastre, I. Ripoll, and A. Crespo, "Analysis of window-constrained execution time systems," *Real-Time Systems*, vol. 35, no. 2, pp. 109–134, 2007.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: an open toolbox for adaptive wcet analysis," in *SEUS*, 2010, pp. 35–46.

[5] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[6] G. Bernat, A. Colin, and S. Petters, *pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems*. University of York, Department of Computer Science, 2003.

[7] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[8] M. Bertogna and S. Baruah, "Limited Preemption EDF Scheduling of Sporadic Task Systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 579–591, 2010.

[9] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," *Real-Time Systems*, vol. 39, no. 1, pp. 5–30, 2008.

[10] D. Broman, "A brief overview of the KTA WCET tool," *arXiv preprint arXiv:1712.05264*, 2017.

[11] A. Burns and A. J. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[12] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[13] Y. Cai and M. C. Kong, "Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems," *Algorithmica*, vol. 15, no. 6, pp. 572–599, 1996.

[14] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, 2007.

[15] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems," in *RTAS*, 2017, pp. 37–48.

[16] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2, pp. 131–181, 1999.

[17] N. Gehani and K. Ramamritham, "Real-time concurrent C: A language for programming dynamic real-time systems," *Real-Time Systems*, vol. 3, no. 4, pp. 377–405, 1991.

[18] B. Guillem, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[19] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *RTSS*, 2006, pp. 57–66.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4*, 2001, pp. 3–14.

[21] A. Hamann, M. Jersak, K. Richter, and R. Ernst, "Design space exploration and system optimization with symta/s - symbolic timing analysis for systems," in *RTSS*, 2004, pp. 469–478.

[22] J.-F. Hermant and L. George, "A C-space sensitivity analysis of Earliest Deadline First scheduling," in *ISoLA*, 2007, pp. 21–33.

[23] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *RTSS*, 1991, pp. 129–139.

[24] J. Knoop, L. Kovács, and J. Zwirchmayr, "r-tubound: Loop bounds for wcet analysis (tool paper)," in *LPAR*, 2012, pp. 435–444.

[25] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *WATERS*, 2015.

[26] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[27] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *RTSS*, 1989, pp. 166–171.

[28] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[29] T. Mirtin, "Realtime programming language pearl-concept and characteristics," in *COMPSAC*, 1978, pp. 301–306.

[30] M. Nasri and B. B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *RTSS*, 2017, pp. 12–23.

[31] S. Natarajan and D. Broman, "Timed C: An Extension to the C Programming Language for Real-Time Systems," in *RTAS*, 2018, pp. 227–239.
[32] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel, "Papabench: a free real-time benchmark," in *WCET*, 2006.
[33] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *POPL*, 1987, p. 188.
[34] C. Ptolemaeus, *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014.
[35] S. Punnekkat, R. Davis, and A. Burns, "Sensitivity analysis of real-time task sets," in *ASIAN*, 1997, pp. 72–82.
[36] R. Racu, A. Hamann, and R. Ernst, "Sensitivity analysis of complex embedded real-time systems," *Real-Time Systems*, vol. 39, no. 1, pp. 31–72, 2008.
[37] R. Racu, M. Jersak, and R. Ernst, "Applying sensitivity analysis in real-time distributed systems," in *RTAS*, 2005, pp. 160–169.
[38] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *RTSS*, 2002, pp. 315–326.
[39] M. Stigge and W. Yi, "Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities," in *Real-Time Syst.*, vol. 51, no. 6, 2015, pp. 639–674.
[40] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "The limited-preemptive feasibility of real-time tasks on uniprocessors," *Real-Time Systems*, vol. 51, no. 3, pp. 247–273, 2015.
[41] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.
[42] K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Syst.*, vol. 6, no. 2, pp. 133–151, 1994.
[43] S. Vestal, "Fixed-priority sensitivity analysis for linear compute time models," *IEEE Transactions on Software Engineering*, vol. 20, pp. 308–317, 1994.
[44] A. Wellings, *Concurrent and real-time programming in Java*. John Wiley & Sons, 2004.
[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 36:1–36:53, May 2008.
[46] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, "An exact schedulability test for non-preemptive self-suspending real-time tasks," in *DATE*, 2019, pp. 1228–1233.
[47] F. Zhang, A. Burns, and S. Baruah, "Sensitivity analysis of the minimum task period for arbitrary deadline real-time systems," in *PRDC*, 2010, pp. 101–108.
[48] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *RTAS*, 2014, pp. 101–110.