Cause-Effect Reaction Latency in Real-Time Systems

Jakaria Abdullah(✉) and Wang Yi(✉)

Uppsala University, Uppsala, Sweden
{jakaria.abdullah,yi}@it.uu.se

Abstract. In embedded real-time systems, a functionality is often implemented as a dataflow chain over a set of communicating tasks. An important requirement in such systems is to restrict the amount of time an input data requires to impact its corresponding output. Such temporal requirements over dataflow chains also known as the end-to-end latency constraints, are well-studied in the context of lock-based blocking inter-task communication. However, lock-based communication does not preserve the functional semantics and complicates latency calculation due to its reliance on response times of the communicating tasks. We propose to use non-blocking inter-task communications to preserve the functional semantics. Unfortunately a naive method to compute the reaction latency by adding worst-case delays between each write-read pair is unsafe for systems with non-blocking communication. In this paper, we study a non-blocking communication protocol. We present an algorithm to compute the exact worst-case delay in a cause-effect chain, which provides a safe estimation of the worst-case cause-effect latency for systems using this protocol for non-blocking communication.

1 Introduction

A simple use case in real-time embedded applications is a dataflow chain where a sampler task samples an input data, passes the data to a controller task for processing and the processed output is used by an actuator task. The specification of the system often includes temporal constraints on such dataflow chains, also known as the end-to-end timing or latency constraints. More specifically, a latency constraint restricts the amount of time required before the input is taken into account by the corresponding output. Latency constraints are important temporal requirements in real-time systems implemented by multiple communicating tasks with different periods. Proving that the implementation of a system satisfies all such requirements is non-trivial but mandatory for safety-critical reasons.

A widely used practice is to design a system using high-level model-based designer tools such as Simulink [1] and verify all the functional requirements using simulation. System design tools like Simulink commonly use specific functional semantics such as the one for Synchronous-Reactive (SR) programming [3] which assumes computation and communication time as zero. When such a

E.-R. Olderog et al. (Eds.): Jonsson Festschrift, LNCS 13030, pp. 41–56, 2021.
https://doi.org/10.1007/978-3-030-91384-7_3

design is implemented on a real platform using preemptive scheduling, the following problems related to inter-task communication arise in preserving any functional semantics [4]:

- Data consistency: If preemption occurs in the middle of an input-output operation, data can get corrupted.
- Deterministic data transfer: Input-output operations of a task may occur at different time points in different jobs depending on the point in the program where preemption takes place. Similarly, non-determinism in execution times creates variation in the timing of the data transfer operations.

The most widely used solution for inter-task communication is lock-based communication. Lock-based communication in a complex embedded system such as the Engine Management System (EMS) [5] makes the design inflexible to change as the computation of end-to-end timing constraints depends on the execution times of all communicating tasks. Besides, lock-based protocols are not designed to preserve any functional semantics [4]. It has been reported that existing state-of-the-art tools for end-to-end timing constraint analysis (in the automotive domain) ignore functional semantics preservation and model transformations are required to ensure data consistency [6].

In a recent work [7], we have shown that non-blocking communication preserving functional semantics is critical for the design of dynamically updatable systems. In this context, we propose to use a wait-free inter-task communication protocol called the DBP [4] in system implementation. DBP preserves functional semantics similar to SR and its correctness does not depend on the execution times of the tasks. This protocol is widely used for designing control systems in the context of synchronous programming. In this work, we study the problem of end-to-end latency computation under this protocol. Specifically, our interest is in computing the reaction latency [8] of an input in a multi-rate dataflow chain either initiated by a periodic or a sporadic task. Here the main challenge lies in the multi-rate nature of the communication where a writer can write at a higher or a lower rate compared to its reader. As a result, input data may not propagate to output or may propagate multiple times.

We show that for tasks using DBP, a worst-case reaction latency computation algorithm that only considers the worst-case delay between releases of each writer-reader pair in a chain provides an unsafe estimation. We also show that the unsafeness of this naive approach originates from the effect of oversampling-undersampling of data in the presence of read-write pairs with non-harmonic periods. We give a safe worst-case reaction latency computation algorithm for the input data that propagates to the output of the chain. This algorithm provides the exact worst-case delay between releases of the first job and the last job in a cause-effect chain.

The rest of the paper is organized as follows. First, in Sect. 2, we review the previous related work on end-to-end timing analysis in the context of multi-rate systems and non-blocking communication. Next in Sect. 3, we give details of the problem and the system model considered in this work. Our proposed latency analysis method is described in Sect. 4 and evaluated in Sect. 5. Finally, we conclude the paper with a summary together with future works in Sect. 6.

2 Related Work

The analysis of latency constraints in multi-rate systems using asynchronous (non-blocking) communication was first studied in the context of synthesizing task parameters [9]. A renewed interest in such analysis stems from an industrial publication [10] where the authors propose a framework to calculate end-to-end latencies in automotive systems supporting the asynchronous communication model. Existing state-of-the-art tools for latency computation such as Symta-S [11] are applicable at the implementation level and mostly based on the availability of lock-based communication. Until recently, state-of-the-art latency analysis techniques such as [12] did not consider the preservation of any functional semantics. Recent research from Bosch [6] emphasized the preservation of model-level functional semantics in end-to-end latency estimation. Similarly, the industrial trend to replace traditional distributed embedded systems with fewer multicore chips increases the potential of semantic preserving non-blocking communication which is difficult to implement in distributed architecture [13].

Non-blocking asynchronous communication for real-time systems is first considered in [14] to meet the freshest value semantics, assuming the data validity time as the worst-case response time of a reader. In [15], an asynchronous protocol that guarantees data consistency with the freshest-value semantics between a writer and multiple readers is presented. This protocol needs hardware-dependent compare and swap operations. This idea of data validity is also used in [16] and [17] to optimize memory use while preserving the freshest value and the SR semantics respectively. These protocols compute a maximum buffer size by upper bounding the number of times the writer can produce new data while a given data is considered valid by at least one reader.

In [18], a double buffer mechanism for one-to-one communication with SR semantics is presented. In the case of uniprocessor systems, given that the code that updates the buffer index variables are executed inside the kernel at task release time, there is no need for a hardware mechanism to ensure atomicity when swapping buffer pointers or comparing state variables. In [4], the Dynamic Buffering Protocol (DBP) is defined for single-writer multiple-reader systems with unit communication delay links, under the assumption that each job finishes before its next release. In [19], the communication scenario presented in [4] is further generalized to handle arbitrary multi-unit communication delays and multiple jobs of a task active at the same time. In [20], multi-task implementation is formulated as an MILP (Mixed Integer Linear Programming) optimization problem which tries to minimize buffer places or total read-write delays in the system to improve control performance. Commercial system design software Simulink [1] provides a wait-free access control mechanism called the Rate Transition (RT) similar to DBP. In the case of communicating tasks with identical phase and harmonic periods, the RT mechanism guarantees data consistency and functional semantics preservation. However, all the above-mentioned works do not consider the computation of latency values in multi-rate dataflow chains.

An alternative non-blocking communication concept called the logical execution time (LET) [21] assumes I/O as time-triggered zero execution time activity which is performed at the release time (read) and the deadline (write) of the task. Although LET preserves time-triggered functional semantics independent of scheduling methods, it increases the delay in data reading as a task can finish computation of data long before the deadline of the current job which affects the end-to-end latency [22].

Prelude [23] is an architecture language intended for the design of multi-rate dependent control systems preserving functional semantics in communication. It supports rate-transition operations similar to Simulink for the needs of multi-rate real-time systems. The communication model assumed in Prelude is causal where a reader is not allowed to start before the completion of its writer. Such causal communication is considered as job-level dependencies where the constraint specifies which job of a writer task needs to finish its execution before a job of the reader task can start. Job-level dependency is used in [24,25] for computing latency constraints at the model-level.

3 Problem Formulation

In this section, we introduce the details of the end-to-end latency problem that we solve and the system model that we use for it. Our considered model is based on automotive software architecture AUTOSAR [8] and end-to-end latencies of complex automotive software like EMS [5].

3.1 Execution Model

An automotive software system consists of multiple software components. The software components that can not be decomposed further are called atomic software components or runnables. In an implementation, a runnable can be implemented as a function that is called whenever required, within the body of an operating system (OS) task. Usually, there are many more runnables in a system than the maximum number of tasks allowed by automotive operating systems. So runnables having the same functional period according to control dynamics are mapped into an OS task with the same period. In the simplest case, one functionality is realized by a single runnable. However, complex functionalities are typically implemented using several communicating runnables which can be distributed on different OS tasks.

We assume runnables of a system S is implemented by a set of n periodic or sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. We denote a periodic task τ_i by a tuple (C_i, T_i, D_i) where WCET $C_i = \sum C_i^j$, T_i is the period and D_i is the relative deadline. Interrupt service routines that are triggered by hardware events are usually modeled as sporadic tasks in the system. In the case of a sporadic task, T_i denotes the minimum inter-arrival time between consecutive jobs. In automotive operating systems only one job of a recurring task can be active at a time. This restriction implies all periodic or sporadic tasks to have deadlines less than or equal to their respective periods.

Tasks are scheduled by the operating system based on the assigned (fixed) priorities. The scheduling policy may be either preemptive or cooperative. Preemptive tasks may always preempt lower priority tasks, while cooperative tasks may preempt a lower priority one only at runnable boundaries. Preemptive tasks are assumed to have a higher priority than any cooperative task.

3.2 Communication Model

Communication between tasks is based on shared memory locations also known as *labels*. A label can be a shared variable allocated in the memory or a register. We assume tasks execute like read-execute-write, where the task reads all the required data at the beginning of execution and writes at the end of its execution.

For accessing a label, AUTOSAR allows two different mechanisms. In explicit or direct access, a runnable directly reads or writes memory location. As a result, data may be overwritten before the reader finishes its reading resulting in data inconsistency. In a more frequently used communication mechanism called implicit access, a task-local copy for data access is created. The copying is performed at the beginning of the job execution and the modified data is written back at the job's termination. Using this mechanism the value of a used label does not change during the runtime of a job and all runnables operate on consistent data. This mechanism is a form of non-blocking communication but does preserve any functional semantics.

In this paper, we use a different non-blocking communication protocol that preserves functional semantics [4] similar to synchronous programming. The principle of synchronous programming is based on the idea of zero time computation and communication. As a result, a data writer task computes and writes its data at the same time when it is released. Then the data reader task can always read the freshest data available at its release time. Here the release time is the time when the job of a task becomes ready for execution.

Let t_i^k represents the release time of the k-th job of task τ_i where $t_i^k \in \mathbb{R}_{\geq 0}$. Now job release times of the task τ_i forms a set $R_i = \{t_i^1, t_i^2, \cdots\}$ where $t_i^k < t_i^{k+1}$. Given time $t \geq 0$, we define $n_i(t)$ to be the maximum index of any job from τ_i that has released before or at t. By definition, $n_i(t) = \sup_k\{k | t_i^k \leq t\}$. We denote x_i^k and y_i^k to be the data that the k-th job of τ_i reads and writes respectively. Now for inter-task communication between writer task τ_i and its reader task τ_j, synchronous semantics assumes:

$$x_j^k = y_i^m, \text{where } m = n_i(t_j^k). \tag{1}$$

As for the case when τ_i has not occurred yet, $m = 0$ and the reader task should read a default value.

In a real execution, tasks do not have zero execution time. In the case of preemptive scheduling, it may be the case that τ_j preempts τ_i before completion. As a result, the τ_i outputs may not be available for τ_j computations. To overcome this problem, a high priority task should read the data written by the job immediately before the latest released writer job as:

$$x_j^k = y_i^m, \text{where } m = max\{0, n_i(t_j^k) - 1\}. \quad (2)$$

For systems where all tasks execute with the same period, synchronous semantics is preserved when the tasks are executed according to their data dependency order. In the case of multi-rate or multi-periodic systems, data may be needed to be communicated between two tasks with different execution rates or periods. The Dynamic Buffering Protocol (DBP) [4] is designed to preserve synchronous semantics in multi-rate multi-task implementation.

We now briefly introduce the DBP protocol in the context of fixed-priority real-time scheduling. The correctness of the protocol is dependent on the following assumptions:

1. The taskset that executes functions communicating using synchronous semantics is schedulable.
2. There is no cyclic communication between tasks without delayed data propagation.
3. All the tasks in the taskset have their relative deadlines constrained by their periods or minimum inter-arrival times.
4. Each pair of communicating tasks should have different fixed priority. In general, the protocol works for any priority assignment policy which assigns a fixed priority to a job during its release time.

To ensure deterministic communication between a writer and its reader tasks, the DBP protocol uses the following rules:

1. A low priority reader job reads the data written by the latest job of its high priority writer released before or with it.
2. A high priority reader job reads the data written by the predecessor job of the latest job of its low priority writer that is released before or with the reader.

The protocol manages the buffer that is written by a writer task and later read by a reader task. Whenever a job is released the kernel or runtime system modifies the pointer to variables which the released task will use for reading and writing. Similarly, when a reader finishes, the runtime marks the used buffer as free. If a reader and a writer are released simultaneously then the pointer fixing function for the writer should execute before the ones for the reader. The code for the original software remains unchanged.

It has been shown that a writer with N readers requires maximum $(N + 2)$ buffer places using the DBP protocol [4]. This bound is intuitive as in the worst-case situation, all the N readers may be still using the different data written by the previous writer jobs when a new writer job is released, thus N buffer places can be in use. The additional two buffer places are to keep the latest and the one-before-latest data to be used by any future arrival of high and low priority readers. Note that here the future release of any reader job means their previous job is finished and one of the previously occupied N buffers is no longer in use.

Automotive software systems sometimes use co-operative scheduling. A low priority co-operative task can block a high priority co-operative task if it starts executing before the release of the second one. However, as long as the tasks are schedulable the DBP protocol does not fail as it is not dependent on the finishing time of the jobs and the protocol ensures a reader job never reads from a writer job that is released after it (Fig. 1).

3.3 Latency Requirements

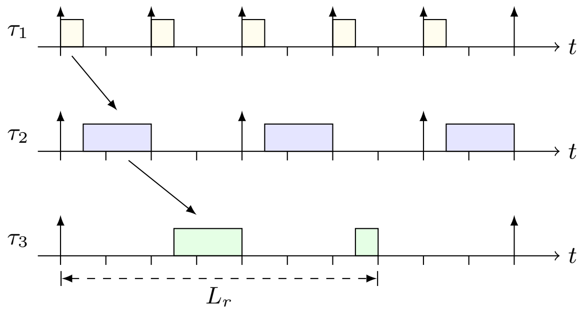

Fig. 1. Reaction latency in a cause-effect chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ comprising three periodic tasks. The arrows indicate flow of data from one task to another one.

In automotive software, a complex functional requirement is often implemented by a chain of runnables where each of these except the first one reads the data written by its predecessor and writes data for its successor. The first runnable in the chain is either released by an event (sporadic) or a periodic polling function. These type of chains are called event chains or cause-effect chains [8]. The runnable that initiates an event chain is called its stimulus and the final runnable in the chain is called its response. A simple example of an event chain is wherein data is sensed by runnable for sensing, passed on to control runnables to compute and finally output of the control runnable is used by the runnables for actuation. A cause-effect chain does not contain any cyclic data dependency [5].

Each of these cause-effect chains is associated with an end-to-end latency requirement. In this work, we are concerned with end-to-end latency from the perspective of a stimulus also known as reaction latency. A reaction latency constraint of L time units to a particular stimulus implies that the first response should occur no later than L time units after that input. As each runnable is mapped into a task and tasks execute in a read-execute-write pattern, we can express reaction latency as the duration between the release time of the task with the first runnable and the completion time of the task with the final runnable.

A cause-effect chain may consists of tasks executing at different rates or periods. Such multi-rate dataflow chains thus often suffer from the effects of undersampling or oversampling of data. Undersampling happens when a slow reader reads from a fast writer and not all data will be read. Oversampling occurs when a fast reader reads from a slow writer and an input data propagates to the output multiple times. With these effects, it is challenging to calculate the reaction latency of multi-rate cause-effect chains due to the following reasons:

– If the chain contains any undersampling effect then the reaction latency calculation should only consider the input data that reaches the output.
– If the chain contains any oversampling effect then the reaction latency calculation should only consider the delay of the first reader job (out of multiple readers that reads the same data) which can propagate data to the next segment of the chain.

Additionally, we have to consider the effect of DBP protocol in reaction latency which preserves the functional semantics mentioned earlier.

3.4 Problem Statement

Given a cause-effect chain $C = \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_N$ with either N synchronously released periodic tasks or a sporadic task τ_1 with $N-1$ synchronously released periodic tasks, we want to calculate worst-case reaction latency of any stimulus of C in uniprocessor where tasks use fixed-priority preemptive scheduling and non-blocking DBP communication protocol.

4 Reaction Latency Estimation

In this section, we present how to compute the worst-case reaction latency of a cause-effect chain in a uniprocessor where tasks are communicating using the non-blocking DBP protocol and scheduled using fixed priority scheduling. As the definition of the worst-case reaction latency, we consider the maximum time that an input data requires to reach the output for the first time by traversing a cause-effect chain. Such an interval starts with the release time of the first job in the chain and finishes with the finishing time of the last job that generates the final output.

4.1 Reaction Latency in Non-blocking Communication

There are two cases of data flow between tasks with different priorities where DBP uses different operations. According to DBP protocol, written data is valid for high priority readers from the release time of the next writer job until the moment before the release of a writer job after that. In the case of low priority readers, the written data is only valid during the interval starting from the writer job release until the moment before its next release. In both cases, the readers released during the defined interval will read the data.

The above cases do not assume anything about periods of the communicating tasks. If the tasks have fixed periods then the oversampling or undersampling effects determine the latency of the propagated data. Note that, the use of a non-blocking communication protocol ensures that the delay of data propagation from a writer to a reader does not depend on the response time of the writer job. Instead, the time distance between the release times of a writer job and its corresponding reader job determines how late the data reaches its final output task. The only response times required to calculate reaction latency are the response times of the jobs of the final task in the chain. So, first we consider a naive way to calculate the worst-case reaction latency of a cause-effect chain $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_N$ as

$$L_{1N} = \sum_{i=1}^{N-1} \Delta_{i \rightarrow i+1} + R_N \tag{3}$$

where $\Delta_{i \rightarrow i+1}$ is the worst-case data propagation delay between tasks τ_i and τ_{i+1}, and R_N is the worst-case response time (WCRT) [26] of the last reader task τ_N. Here the data propagation delay means the time distance between release times of a writer job and its corresponding reader job. R_N can be calculated by the smallest value of R_N that satisfies the recursive equation

$$R_{N_i} = C_r + \sum_{j \in hp(i)} \left\lceil \frac{R_{N_i}}{T_j} \right\rceil \cdot C_j. \tag{4}$$

Note that due to the effect of undersampling a data may not reach the output of the chain. Similarly, due to the effect of oversampling, the job that propagates the data may not be the first job that reads this data. As a result, *Algorithm 1* may not give a safe overapproximation of the worst-case reaction latency of the input data that reaches the output.

To show the problem of adding local worst-cases in delay computation, we use a simple cause-effect chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ with three periodic tasks as shown in Fig. 3. We assume $prior(\tau_3) < prio(\tau_1) < prio(\tau_2)$. As we see from the Figure, due to the combination of low to high (oversampling) and high to low (undersampling) data propagation, the first job of τ_2 that reads data from τ_1 is not propagating data to the next reader task τ_3. Note that such an effect can only happen when the writer and its reader task have non-harmonic periods. This makes *Algorithm 1* unsafe as it assumes that the first reader of any data is always propagating it (Fig. 2).

We observe that if a cause-effect chain contains data exchange between a fast writer and a slow reader then many of the writer jobs will not be able to propagate data to its reader task. In the simplest case, we consider the cause-effect chain consists of only two tasks executing at different rates or periods. In a two task event chain, we have a writer task $\tau_w = (C_w, T_w, D_w)$ and a reader task $\tau_r = (C_r, T_r, D_r)$. According to DBP protocol, a reader job only reads data from two types of writer jobs released before or with it. The first type used in high to low priority communication is the latest writer job that is released before or with the release of the reader. If the reader job is released at time t then the maximum index of a writer job released in the interval $[0, t]$ is $\lfloor \frac{t}{T_w} \rfloor$. In case of low to high priority communication, the relevant job is the one released immediately before the latest writer job. The index of such a job is $\lfloor \frac{t}{T_w} \rfloor - 1$.

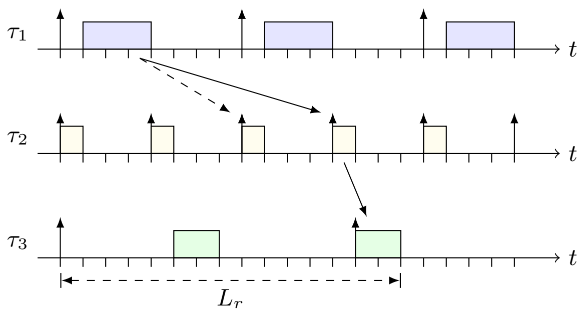

Fig. 2. The effect of oversampling in reaction latency of a cause-effect chain. The arrows indicate dataflow between tasks. The dashed arrow shows the first read that is not propagated.

Based on the observations regarding data misses in a cause-effect chain, we have an algorithm to compute delays of all data that reach from input to output. The algorithm shown in Fig. 3 starts from the jobs of the final task in the chain and computes indexes of the writer jobs that are propagating each of the data. All possible job release combinations of a synchronous periodic taskset are present in the hyperperiod of the taskset which is equal to the Least Common Multiple (LCM) of all the periods in it. As DBP protocol also requires reading data from the previous period of the writer, our algorithm needs to check job release propagation in an interval of at least twice the length of the hyperperiod. This requirement is necessary because the first two tasks in a chain can have low to high priority communication. The algorithm works like this, we use a two-dimensional matrix M where each row represents a task in the chain and each column represents a data propagation path. Starting from the last row with job indexes of the output task, we calculate the indexes of the writer jobs using $\lfloor \frac{t}{T_w} \rfloor - 1$ or $\lfloor \frac{t}{T_w} \rfloor$ based on priorities of the reader and writer tasks where t is the release time of the reader job. A special case happens when the calculated index becomes negative. The origin of this negative value is low to high communication in DBP. In it, the initial reader jobs are reading a default value due to the absence of propagated data. We mark these indexes with -1 in the matrix. In the computed matrix, a non-negative value in the item $M[i][j]$ indicates the job index of the job of task τ_{i-1} that propagates data to $(j-1)$-th job of the final output task in the chain.

The output matrix of the algorithm in Fig. 3 captures information of all the jobs that are included in any data propagation path of a cause-effect chain that reaches the output. To get the maximum delay in any such path from M, we use the algorithm in Fig. 4. This algorithm checks for each non-negative job index of the input task τ_0, the corresponding release distance of its reader job from the output task τ_{N-1}. As we are only concerned with the first output of data, the algorithm skips the input data that are read multiple times. The maximum value among these distances is the maximum delay a data suffers reaching from

```
1: $C$: chain $\{\tau_0, \tau_1, \ldots, \tau_{N-1}\}$ of $N$ tasks
2: $M$: $N \times (2 \cdot P + 1)$ zero matrix where $P = \frac{LCM(C)}{T_N}$
3: procedure End-to-end$(T, M)$
4:     $c_i \leftarrow 0$
5:     for $j = 0$ to $2 \cdot P$ do
6:         $M[N-1][j] \leftarrow c_i$
7:         $c_i \leftarrow c_i + 1$
8:     for $i = N-1$ to $1$ do
9:         for $j = 0$ to $2 \cdot P$ do
10:            $t \leftarrow M[i][j] \cdot T_i$
11:            if $prio(\tau_i) > prio(\tau_{i-1})$ then
12:                $w \leftarrow \lfloor \frac{t}{T_{i-1}} \rfloor - 1$
13:            else
14:                $w \leftarrow \lfloor \frac{t}{T_{i-1}} \rfloor$
15:            if $(w \cdot T_{i-1} \geq 0)$ then
16:                $M[i-1][j] \leftarrow w$
17:            else
18:                $M[i-1][j] \leftarrow -1$
19:    return $M$
```

Fig. 3. Algorithm for computing delays in all cause-effect chains where data reaches the output

```
1: $C$: chain $\{\tau_0, \tau_1, \ldots, \tau_{N-1}\}$ of $N$ tasks
2: $M$: $N \times (P + 1)$ matrix from algorithm in Figure 3.
3: procedure Find-Max-Chain$(T, M)$
4:     $Max \leftarrow 0$
5:     $V \leftarrow 0$
6:     $Prev \leftarrow -1$
7:     for $j = 0$ to $2 \cdot P$ do
8:         if $(M[0][j] \geq 0) \wedge (M[0][j] \neq Prev)$ then
9:             $V \leftarrow M[[N-1][j] \cdot T_{N-1} - M[0][j] \cdot T_0$
10:            $Prev \leftarrow M[0][j]$
11:            if $V > Max$ then
12:                $Max \leftarrow V$
13:    return $Max$
```

Fig. 4. Algorithm for finding maximum delay in cause-effect chains.

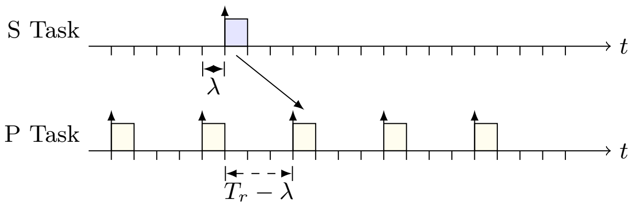

Fig. 5. Worst-case situation in communication between a high priority sporadic writer task and its periodic reader. The sporadic task is released $\lambda > 0$ time units after the previous reader job.

the input to the output. Finally, a safe upper bound of the worst-case latency can be calculated by adding the WCRT of the final task with the calculated delay. We denote this latency computation method as *Algorithm 2*.

Note that the complexity of *Algorithm 2* is linear in the size of the resulting matrix M. This means our algorithm has a time-complexity linear in the size of the hyperperiod of the tasks in a cause-effect chain. The hyperperiod is in the worst-case exponential with respect to the number of tasks of the chain. However, this worst-case happens when all the communicating tasks have co-prime periods which is a rare case in practice.

4.2 Data Propagation Delay Between Sporadic Input and Synchronous Periodic Output Tasks

Now we consider a special type of cause effect chain where the first task in the chain is released sporadically with a minimum inter-arrival time. Similar to the analysis of periodic tasks, we first consider the simple case of a sporadic writer τ_w and a periodic reader task τ_r. As τ_w is sporadic, T_w is the minimum inter-arrival time between two writer jobs. We assume a sporadic writer task is always assigned higher priority than its reader task. This is reasonable considering the fact that sporadic tasks are event-triggered for capturing input events (Fig. 5).

For $T_w > T_r$, the reader task will always read the latest data written by the writer. Note that for $T_w \leq T_r$, multiple sporadic writer jobs can release between releases of two consecutive reader jobs. As the sporadic writer job with new data can arrive indefinitely later after T_w, even the slowest periodic reader task will oversample in the absence of new writer jobs. In that case, the delay suffered by the first reader job determines the reaction latency of the sporadic writer. In the worst-case, the latest sporadic writer releases immediately after the release of a reader job to maximize the delay for the next release of the reader job that will read the data for the first time. Hence the maximum delay between releases of the periodic reader and sporadic writer is

$$S_{hl}(\tau_w, \tau_r) = T_r - \lambda \qquad (5)$$

where $\lambda > 0$ is the smallest granularity of time by which an operating system can separate two consecutive job release events.

Suppose we have a cause-effect chain $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_N$ where τ_1 is sporadic and rest of the tasks are periodic. We calculate the maximum delay D_{2N} for the periodic part of the chain (τ_2 to τ_N) using either *Algorithm 1* or *Algorithm 2*. If $prio(\tau_1) > prio(\tau_2)$, then using Eq. 5 we can calculate a safe upper bound on worst-case reaction latency as

$$L^S_{1N} = T_2 - \lambda + D_{2N} + R_N. \tag{6}$$

5 Evaluation

For evaluation, we implemented the algorithms described in Sect. 4 using Python programming language. We consider cause-effect chains from the Bosch case study of an EMS [5]. The case study is for a multi-core platform with a global memory and local scratchpads. Interestingly, each core can access all the scratchpads via a crossbar connection. Although the chains in the case study allow placing tasks in different cores, we consider all the tasks of a chain to execute in the same processor. This is reasonable as we ignore memory access overhead and the only effect of placing tasks in different core in our analysis is the change of WCRT values of the final output task. In the case study, all the periodic tasks have periods in milliseconds (ms) such as $1, 2, 10, 20, 50, 100, 200, 500$ and 1000. Sporadic tasks have their inter-arrival times specified in microseconds. All the tasks in the case study are assigned unique priorities using rate-monotonic policy [27]. These priorities are positive integers where a large value means a high priority.

In the evaluation, we used *Algorithm 1* and *Algorithm 2*. We want to highlight the unsafeness of *Algorithm 1* and use the following result from [28]:

$$\Delta_{i \rightarrow i+1} = \begin{cases} T_i + \min(T_i, T_{i+1}) - gcd(T_i, T_{i+1}), & \text{if } \pi_i < \pi_{i+1} \\ \min(T_i, T_{i+1}) - gcd(T_i, T_{i+1}), & \text{if } \pi_i > \pi_{i+1} \end{cases} \tag{7}$$

where π_i represents priority of τ_i.

Figure 6 shows reaction latency computation of three chains with periodic tasks. Here the first chain is from the case study [5]. Note that, as each pair of tasks in the considered chains has harmonic periods, both of our algorithms computed the same latency values. For the third chain, the reaction latency is the WCRT of the output task because all the write-read pairs have high-to-low data transfer.

Next, we compute reaction latency of cause-effect chain with sporadic stimulus where we consider λ is $1\,\mu$s. We use two such chains where minimum inter-arrival times of sporadic input tasks are specified in microseconds. For calculation, we convert periods of the periodic tasks into microseconds. Figure 7 shows the latency of two such chains where both of our algorithms give identical latency due to the harmonic periods of the periodic tasks. Here the first sporadic chain

Chain Periods	Priority	Latency Alg1	Latency Alg2
$100 \rightarrow 10 \rightarrow 2$	$[1, 2, 3]$	$110 + R_2$	$110 + R_2$
$20 \rightarrow 10 \rightarrow 100$	$[2, 3, 1]$	$20 + R_{100}$	$20 + R_{100}$
$10 \rightarrow 20 \rightarrow 50$	$[3, 2, 1]$	R_{50}	R_{50}

Fig. 6. Reaction latency computations using Algorithm 1 and 2 where R_i is the WCRT of task with period i.

Chain Periods	Priority	Latency
$800 \rightarrow 2000 \rightarrow 50000$	$[3, 2, 1]$	$1999 + R_{50000}$
$6660 \rightarrow 10000 \rightarrow 20000$	$[3, 2, 1]$	$9999 + R_{20000}$

Fig. 7. Reaction latency computations for chain with sporadic task where R_i is the WCRT of task with period i.

Chain Periods	Priority	Latency Alg1	Latency Alg2
$10 \rightarrow 35 \rightarrow 50$	$[3, 2, 1]$	$35 + R_{50}$	$30 + R_{50}$
$7 \rightarrow 1 \rightarrow 100$	$[2, 3, 1]$	$7 + R_{100}$	$13 + R_{100}$
$100 \rightarrow 3 \rightarrow 8$	$[1, 3, 2]$	$104 + R_8$	$108 + R_8$

Fig. 8. Reaction latency computations with non-harmonic periods using Algorithm 1 and 2 where R_i is the WCRT of task with period i.

is from [5] and the second chain assumes an angle-synchronous task as sporadic input.

Finally, Fig. 8 shows how reaction latencies computed by both algorithms differ in the presence of non-harmonic periods between communicating tasks. We used three chains each consisting of three periodic tasks where all write-read pairs do not have harmonic periods. We see for the second and the third chain of Fig. 8, *Algorithm 1* computes unsafe lower delays compared to *Algorithm 2*. As the differences in calculated delays are not so large, it is intuitive that the usefulness of *Algorithm 2* will be more evident in longer chains with more non-harmonic read-write pairs. However, the maximum number of tasks in multi-rate cause-effect chains is three in the case study [5].

6 Conclusion

In this paper, we have studied the problem of estimating the worst-case reaction latency of a cause-effect chain in the multi-rate real-time system that uses non-blocking inter-task communication. We have shown that any naive estimation algorithm that combines the worst-case data propagation delays of each write-read pair is unsafe. We provide an algorithm to compute the exact worst-case data propagation delay between releases of a stimulus and its response in cause-effect chains. Our algorithm does not depend on the response times of the data writer jobs and only assumes the system to be schedulable. An evaluation based on a realistic system [5] shows that our algorithm is able to remove the unsafeness of the naive approach.

As future work, we want to provide a more efficient algorithm for reaction latency computation that can construct the global worst-case situation without enumerating paths of all reachable data. We will extend this work for multi-core platforms and will evaluate the overheads of the considered non-blocking communication protocol. Another interesting direction in latency computation is to consider age latencies [8]. Age latencies are important for control performance but these are more relevant in the system which can tolerate deadline misses.

Finally, we would like to thank authors Bengt Jonsson and Hans Hansson for their seminal work [2] on temporal logic to check timing properties in probabilistic chains. To the best of our knowledge, their work is one of the earliest known contributions to check properties similar to the reaction latency. The problem is still relevant in different settings and their pioneering work continues to inspire us.

References

1. Simulink user's guide: the MathWorks. Natick, MA, USA (2016)
2. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comput. **6**(5), 512–535 (1994)
3. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Sci. Comput. Program. **48**(1), 21–42 (2003)
4. Caspi, P., Scaife, N., Sofronis, C., Tripakis, S.: Semantics-preserving multitask implmentation of synchronous programs. ACM Trans. Embed. Comput. Syst. **7**(2), 15:1–15:40 (2008)
5. Kramer, S., Ziegenbein, D., Hamann, A.: Real world automotive benchmark for free. In: 6th International Workshop on Tools and Methodologies for Embedded and Real-time Systems at ECRTS 15, July 2015
6. Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F.: Communication centric design in complex automotive embedded systems. In: 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, pp. 10:1–10:20 (2017)
7. Yi, W.: Towards customizable cps: composability, efficiency and predictability. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 3–15. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_1
8. Specification of timing extensions, autosar.org. https://www.autosar.org
9. Gerber, R., Hong, S., Saksena, M.: Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In: Proceedings of Real-Time Systems Symposium, pp. 192–203 (1994)
10. Feiertag, N., Richter, K., Nordlander, J., Jonsson, J.: A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In: Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (co-located with RTSS 2008) (2008)
11. Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System level performance analysis - the symta/s approach. IEE Proc. Comput. Digit. Tech. **152**(2), 148–166 (2005)

12. Schlatow, J., Ernst, R.: Response-time analysis for task chains in communicating threads. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–10 (2016)
13. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Natale, M.D.: Implementing synchronous models on loosely time triggered architectures. IEEE Trans. Comput. **57**(10), 1300–1314 (2008)
14. Kopetz, H., Reisinger, J.: The non-blocking write protocol NBW: a solution to a real-time synchronization problem. In: Proceedings Real-Time Systems Symposium, pp. 131–137 (1993)
15. Chen, J., Burns, A.: A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. York University, Technical report (1997)
16. Huang, H., Pillai, P., Shin, K.G.: Improving wait-free algorithms for interprocess communication in embedded real-time systems. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, pp. 303–316 (2002)
17. Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A.: Efficient embedded software design with synchronous models. In: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 187–190 (2005)
18. Scaife, N., Caspi, P.: Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems, pp. 119–126 (2004)
19. Di Natale, M., Wang, G., Vincentelli, A.S.: Optimizing the implementation of communication in synchronous reactive models. In: IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 169–179 (2008)
20. Natale, M.D., Guo, L., Zeng, H., Sangiovanni-Vincentelli, A.: Synthesis of multitask implementations of simulink models with minimum delays. IEEE Trans. Ind. Inf. **6**(4), 637–651 (2010)
21. Kirsch, C.M., Sokolova, A.: The logical execution time paradigm. In: Chakraborty, S., Eberspächer, J. (eds.) Advances in Real-Time Systems. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24349-3_5
22. Matic, S., Henzinger, T.A.: Trading end-to-end latency for composability. In: 26th IEEE International Real-Time Systems Symposium (RTSS 2005), pp. 12–110 (2005)
23. Pagetti, C., Forget, J., Boniol, F., Cordovilla, M., Lesens, D.: Multi-task implementation of multi-periodic synchronous programs. Discret. Event Dyn. Syst. **21**(3), 307–338 (2011)
24. Becker, M., Dasari, D., Mubeen, S., Behnam, M., Nolte, T.: End-to-end timing analysis of cause-effect chains in automotive embedded systems. J. Syst. Archit. **80**, 104–113 (2017)
25. Forget, J., Boniol, F., Pagetti, C.: Verifying end-to-end real-time constraints on multi-periodic models. In: 22nd IEEE International Conference on Emerging Technologies and Factory Automation, pp. 1–8 (2017)
26. Joseph, M., Pandya, P.: Finding response times in a real-time system. Comput. J. **29**(5), 390–395 (1986)
27. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
28. Abdullah, J., Dai, G., Yi, W.: Worst-case cause-effect reaction latency in systems with non-blocking communication. In: DATE, pp. 1625–1630 (2019)