Analysis of a Chain Starting with a Time-Driven Component

M.A.Weffers-Albu, J.J.Lukkien, P.D.V. van der Stok
Technische Universiteit Eindhoven, Philips Research Laboratories Eindhoven
{m.a.albu, j.j.lukkien}@tue.nl, peter.van.der.stok@philips.com

Abstract

In this article we provide a model for the dynamic behavior of media processing chains of software components communicating via bounded buffers. Components can have variable computation times and the first component in the chain has a periodic behavior. The aim is to find the overall behavior of a chain from which performance parameters (such as start time and response time of individual tasks, chain end-to-end response time, number of context switches and resource utilization) follow. This behavior is characterized in terms of the elementary actions of the tasks that make up the chain, and the associated schedule function. Knowing the behaviour of the chain at design time allows the optimization of resources, which is achieved by minimizing context switching overhead and buffer sizes. Additionally we show what are the conditions to be imposed at design time such that the chain satisfies its real-time constraints that influence directly the quality of service provided.

1 Introduction

We consider the problem of processing a media stream by a system consisting of a chain of given off-the-shelf software components, executed on a scarce-resource embedded platform. Each component corresponds to a task, and the communication between tasks is buffered. The essential requirement on the physical platform is cost-effectiveness, and the requirement on the system is robustness. These requirements lead to minimizing the resources made available to the system to the limit that it remains robust.

In the context of our work the criteria for robustness for a system are meeting its real time constraints. The real-time constraints come from the fact that media processing systems must capture and display audio/video information at a certain rate in order to avoid audio/video artefacts. This implies that the degree in which the real-time constraints are met directly influences the quality of service provided by the system.

To ensure that the timing constraints are met, the chain is provided with a guaranteed resource budget [9, 10]. Within the chain, the tasks are scheduled using fixed priority scheduling. Due to the dependencies between the tasks, and their different behaviors, it is difficult to predict the behavior of the chain. Hence, it is difficult to determine the minimum needed resource budget, to predict chain response time, to minimize buffer sizes and context switch overhead, and to reason about chain composition.

The current practice in the domain of media processing systems lacks a theoretical underpinning that helps designers and developers beyond intuition and experience. Such a theory is also needed to control the chain behavior at design time, to make sure timing requirements are met even in overload situations.

1.1 Contribution and paper organization

Our research aims at providing an underlying theory that helps engineers to reason rigorously about system behavior and associated resource needs. It starts from the experimental observation that, under

certain conditions, a media processing chain assumes a repetitive behavior, the stable phase, after a finite initial phase. Starting from this observation we are building a theoretical model for the execution of streaming chains in media processing systems. Our general strategy is to analyze streaming systems in an incremental manner starting from a simple theoretical case, to realistic streaming chains that include branching and more complex types of components. In [12] we presented the analysis of an initial case, of a linear chain executing in a cooperative environment. In this article we continue that analysis by presenting the case of a linear chain in which the first component has periodic behaviour (such as the video digitizer component in a streaming chain).

Our approach allows us to calculate the execution order of the components in a chain, expressed as a trace of actions taken by each component. We formally prove that the behavior of the chain can be expressed as a unique trace, which, under specific conditions imposed at chain design time, assumes a repetitive pattern after a finite prefix. The trace is completely determined by the individual traces of the components, the computation times of the component actions, the topology of the chain, the capacities of the communication buffers, and the static priorities of the components. Given the computation times for each action in the trace, the associated schedule can be derived. The unique trace of actions and the schedule prove an excellent starting point for further analysis. Start times and response times of the individual components and the complete chain are immediately available. The number of context switches, and the position of the context switches in the component traces, which is an indicator for their overhead cost, can be extracted from the trace. Also given the individual traces of the components and the channel constraints, we calculate the minimum necessary and sufficient capacities for each buffer in the chain.

This paper is organized as follows. In section 2 we discuss related work. Section 3 gives insight about the basic notions of trace theory and sets the general approach for our work. Section 4 presents the architecture style of our system and the types of components we study. Section 5 presents the process we use to derive the unique trace and schedule that specify the chain behaviour, properties of this trace, and best practices for design with respect to optimization of resources and meeting timing constraints. We answer practical questions regarding the calculation of chain response time, buffer capacities and number of context switches. We show as well how the latter two can be optimized. We end our presentation with conclusions in section 6.

2 Related work

Several attempts have been made to analyze message passing, streaming systems. Closely related work [6] considers also an execution model for video streaming chains inspired by *TriMedia Streaming Software Architecture*. The article [6] presents an analysis method allowing the calculation of the worst-case response time of multiple video streaming chains based on the canonical form of the chains. The assumptions adopted are that tasks have fixed execution times, tasks are allowed to have equal priorities and the overhead introduced by context switches is ignored. Their approach is based on the response time analysis for tasks with deadlines beyond the periods [5].

Klein et al. [8] apply fixed-priority response time analysis to message-passing systems. The system is modeled in terms of events and event responses. Message handlers create new events if outgoing messages are sent at a different rate than incoming messages. Tasks are modeled as shared resources. The processing of a message by a task is modeled as an atomic action on the shared resource. This leads to the response-time analysis of a set of independent event responses with atomic access to shared resources.

Goddard [4] studies the real-time properties of PGM dataflow graphs, which closely resemble our media processing graphs. Given a periodic input and the dataflow attributes of the graph, exact node execution rates are determined for all nodes. This is very similar to the approach presented in [8]. The periodic tasks corresponding to each node are then scheduled using a preemptive EDF algorithm. For this implementation of the graph, the author shows how to bound the response time of the graph and the buffer

requirements. Both approaches consider complete task sets scheduled by a single scheduling algorithm, and are limited to task sets with deadlines equal to the period, i.e. without self-interference.
Each of these attempts provided valuable insights, but none of them helps engineers to reason in the detail we need about system behavior and associated resource needs. By focusing on the behavior of the chain, our work bears a close resemblance on work done by Gerber et al [3] on compiler support for real-time programs. One idea pursued in that work is to make sure that only relevant actions are performed between trigger and result-delivery, and that housekeeping tasks are delayed till after completion of the real-time part. In our approach, similar results are obtained by appropriate priority assignments to components.

3 Model

We use a simple intuitive syntax for the program text of the components using repetition ('while'), selection ('if'), sequential composition (';') and basic statements (communication and computation actions). The set of basic statements of a component C is called its alphabet $A(C)$. Alphabets of different components are disjoint ($A(C_i) \cap A(C_j) = \emptyset \Leftrightarrow i \neq j$). The semantics are defined as the set of sequences that correspond to the possible execution sequences of these actions according to the program. These sequences are called the traces of the component. An example of a component with corresponding traceset is $C : \{a;b;c;d\}$ (perform the four actions sequentially) with traceset $Tr(C) : \{(abcd)\}$. A trace is a finite or infinite sequence of symbols. Traces s and t can be combined by concatenation to st; if s is infinite this concatenation is just s. Concatenation is generalized to sets of traces in the obvious way. The length of a trace t is written as $|t|$. For trace t, $Pref(t)$ denotes the set of all prefixes of t. Besides this regular behaviour we introduce two extensions: infinite repetition and parallel composition. The infinite repetition corresponds to a set of infinite traces. For example, $C : while(true)\ do\ \{a;b\}$ has traceset $Tr(C) : \{(ab)^{\omega}\}$ where ω specifies infinite repetition. A realistic component example common to the video processing domain would be a sharpness enhancement component. Such a component receives as input a decoded video frame, performs a sharpness enhancement algorithm on this input, and finally sends as output the processed frame. This scenario is repeated infinitely many times. The syntax of the component is

$$C : while(true)\ do \quad \{receive(frame); enhance(frame); send(frame)\}.$$

The traceset is similar as for the component above. Parallel composition of two components is denoted by a parallel bar: $||$. The semantics for one processor is defined as the interleaving of the tracesets. For example $C_0||C_1$, with $C_0 : \{a;b\}$ and $C_1 : \{c;d\}$ has traceset: $\{(abcd),(acbd),(acdb),(cadb),(cdab),(cabd)\}$. We define the projection of a trace t on a certain alphabet A, denoted by $t \uparrow A$, as the trace obtained from t by removing all symbols not in A while maintaining the order given in t. For trace t and symbol a we define the counting operator # as follows:

$$\#(t,a) = |t \uparrow a|$$

Informally, $\#(t,a)$ denotes the number of occurrences of a in t. Also the meaning of *a in t*, ($a \in A$ and t a trace), is that action a occurs in trace t, that is $\#(t,a) \geq 1$.

3.1 States, invariants and channels

A traceset associated with a component represents all possible complete executions. A machine being in the middle of such an execution has only executed a prefix of such a trace. We associate such a prefix with a state during execution. Hence, the set of states during execution is characterized by the prefixes of the traceset, the *prefix closure*. For a traceset T we denote this set of states by $St(T)$.
Properties that are true for all elements of a traceset are called invariants. For example, with *C: while (true) do* $\{a;b\}$ we have

$$I : 0 \leq \#(t,a) - \#(t,b) \leq 1$$

for all states t in $St(Tr(C))$. This invariant merely states that actions a and b alternate in every (partial) execution. In such an invariant we drop the trace argument from the function and simply say that

$$0 \leq \#a - \#b \leq 1$$

is an invariant of C (or of $Tr(C)$).

Invariant I above is an example of a synchronization condition. Actions a and b are in this example synchronized by virtue of the program syntax. We call such an invariant a *topology invariant*. However, instead of looking at invariants that the traceset already has, we can also *impose* invariants. This represents limitations on the execution of the atomic actions. These imposed invariants then lead to a restriction to the subset of traces and corresponding states for which the invariants hold.

As an example, consider once more the following:

$C_0||C_1$, with C_0*: while (true) do* $\{a;b\}$ and C_1*: while (true) do* $\{c;d\}$.

$Tr(C_0||C_1) = \{s : s \quad consists \quad of \quad elements \quad of\ A(C_0) \cup A(C_1) \wedge s \uparrow A(C_0) \in Tr(C_0) \quad \wedge$
$s \uparrow A(C_1) \in Tr(C_1)\}$.

We now decide that action a represents a *send* action and d a *receive* action on an unbounded channel. This interpretation leads to imposing the invariant $0 \leq \#a - \#d$. This means that in the above set certain traces are ruled out as a behaviour. (Notice that the invariant must hold for all states derived from this set). Alternatively, we may decide that a and d form a synchronous channel as in CSP. This is captured in the invariant that a and d always follow each other immediately in the trace while their order is unimportant. This limits the traceset again; notice that this effectively orders c and b as well. As another example, assume that we assign C_0 a higher priority than C_1. This translates into the invariant that no C_1 actions can precede C_0 actions.

Send and *receive* actions via a channel c are denoted as $c!$ and $c?$ respectively like in CSP [7]. However, in contrast to CSP we use bounded channels by imposing the invariant

$$0 \leq \#c! - \#c? \leq Cap(c) \tag{1}$$

with $Cap(c) > 0$ being the channel capacity.

We denote the number of elements in a channel c with $L(c)$. We note that $L(c) = \#c! - \#c?$. Therefore (1) can be rewritten as: $0 \leq L(c) \leq Cap(c)$.

The introduction of constraints on the interleaved behaviour also leads to the notion of blocking. Consider a constrained traceset T and a particular state s of the system, i.e., an element of the prefix closure $St(T)$. Suppose also that the traceset is a result of a parallel composition $C_0||C_1$. Any action a in $A(C_0)$ such that $sa \uparrow A(C_0)$ is a state of $Tr(C_0)$ and sa is a state of T is called a *ready* action of C_0 in state s. If sa is not a state of T it is apparently not possible to execute a in the constrained set. We say that C_0 is blocked at a in state s of T, denoted as $'C_0\ \boldsymbol{b}\ a$[in s of $T]'$. In most cases both s and T are clear from the context and then we leave them out. When T is given, then in any state s we can divide the set of components into *blocked* components ($B(s)$) and *ready-to-run* components ($RR(s)$).

We close this section with adding a few more concepts relevant to our model.

We define $Comp$ as a function taking as argument an action and returning the component with the alphabet to which action a_i belongs. As an example for an action $a_i \in A(C_i)$, $Comp(a_i) = C_i$.

Furthermore, considering a trace t written as $t = s_0 abs_1$. If $Comp(a) \neq Comp(b)$, then we say that a *context switch* occurs between $Comp(a)$ and $Comp(b)$ in state $s_0 a$ of t.

Finally, we define the *number of context switches (NCS)* function taking as argument a trace from a traceset T, and returning the number of context switches occurring in the trace:

$$NCS : T \rightarrow \mathbf{N}, NCS(\varepsilon) = 0, NCS(a) = 0,$$
$$NCS(tab) = \begin{cases} NCS(ta) & if \quad Comp(a) = Comp(b) \\ NCS(ta) + 1 & otherwise \end{cases}$$

3.2 Approach

We consider a system consisting of a parallel composition of communicating components. The corresponding traceset is limited to those traces that satisfy the channel properties (1) for all channels. We

call this the *channel-consistent* traces. On top of this set we impose priorities. This results in just a single trace for the system, depending on the priority assignment. Characterizing precisely this trace is one of our targets. Besides this we add timing properties, which we can analyze as well. In short, we can analyze the system in terms of time and behaviour as a function of the choice of the atomic action order in the components, the channel properties (e.g., the capacity), the timing assignment, and the priority assignment.

4 A Streaming Pipeline, System Architecture

We focus on systems consisting of a collection of communicating components connected in a pipelined fashion (Pipes and Filters architecture style [1]). An instance of this architecture style, the TriMedia Streaming Software Architecture (TSSA)[2] provides a framework for the development of real time audio-video streaming systems executing on a single TriMedia processor. A media processing system is described as a graph in which the nodes are software components that process data, and the edges are channels (finite FIFO queues) that transport the data stream in packets from one component to the next. A simple example of such a chain is presented in Figure 1.

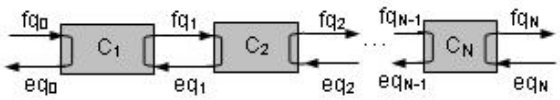

Figure 1. Chain of components

Every connection between two components is implemented by two queues. One queue (*forward queue*) carries full packets containing the data to be sent from one component to the next, while the second queue (*backward queue*) returns empty packets to the sender component to recycle packet memory. The empty packets are returned to signal that the data has been received properly and that the associated memory may be reused. The capacity of fq_i is equal to the capacity of eq_i. Each component $C_i (1 \leq i \leq N)$ has an associated task to which a unique fixed priority is assigned and the tasks execute as long as input is provided. The initial situation of the chain is that all forward queues (except fq_0) are empty and that all backward queues (except eq_0) are filled to their full capacity. This is expressed as follows:

$$L(fq_i) = 0 \wedge L(eq_i) = Cap(eq_i), \quad 1 \leq i \leq N.$$

The behavior of C_i, $2 \leq i \leq N$, is the following (Figure 2): the component receives 1 full packet (p) from the input forward queue ($fq_{i-1}?$) ❶, then receives 1 empty packet (q) from the input backward queue ($eq_i?$) ❷, performs the processing (c_i) ❸, recycles the input packet p from fq_{i-1} by sending it in the output backward queue ($eq_{i-1}!$) ❹ and finally, the result of processing (q) is sent in the output forward queue ($fq_i!$) ❺.

The syntax for the execution of a component C_i, $1 \leq i \leq N$, and the traceset are:

$C_i : while(true)\ do$

$\{receive(fq_{i-1}, p); receive(eq_i, q);$

$process\_fct\_c_i(p,q);$

$send(eq_{i-1}, p); send(fq_i, q);\}$

with traceset: $Tr(C_i) : \{(fq_{i-1}?\ eq_i?\ c_i\ eq_{i-1}!\ fq_i!)^{\omega}\}$, i.e. an infinite sequence of actions. We call these components *data-driven* components. From the syntax of C_i, $2 \leq i \leq N$ we obtain the following topology invariants:

$$0 \leq \#fq_{i-1}? - \#eq_{i-1}! \leq 1 \tag{2}$$

$$0 \leq \#eq_i? - \#fq_i! \leq 1 \tag{3}$$

The first component in the chain (C_1) has similar behavior as the rest with respect to the operations on the queues. However this component has periodic behavior meaning that it is supposed to receive and

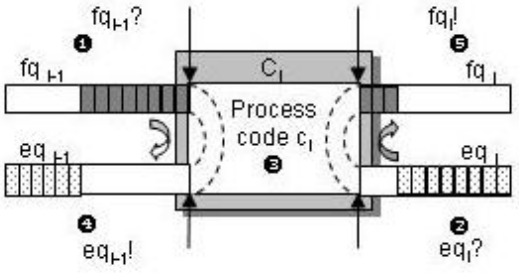

Figure 2. A basic streaming component

process a new packet from fq_0 each T_1 time interval. At the programming level we add an action called $delay\_until(i*T_1)$ where $i*T_1$ refers to an absolute time. The informal interpretation of $delay\_until(i*T_1)$ is that it delays resuming the execution of component C_1 until time is at least $i*T_1$. The computation time of the $delay\_until$ action is 0.

An example from practice of such a component is the video digitizer component. The video digitizer captures periodically video images (frames/fields) via a video camera. These images are passed on to the other components down the chain most commonly to be displayed on a TV screen, or to be encoded and saved on a storage facility.

In general, the video digitizer component will attempt obtaining an empty packet where the captured data will be stored. After capturing an image (field), the digitizer will attempt to acquire another empty packet. If successful, it will send out the packet with the recently acquired image to the next component in the chain. If acquiring an empty packet fails, the digitizer will simply use the packet which it has in possession to store the next image. In this case the already filled packet will not be sent out and will instead be overwritten with the new incoming digitized data. This means that the previously captured image will never be sent down the chain for an eventual display. For perceived QoS purposes it is essential that C_1 always can acquire a new empty packet in eq_1 every T_1 so that it can send periodically captured images to the rest of the components in the chain.

We include below the program that specifies the complete behaviour of C_1:

C_1 :
$i := 1;$
$receive(fq_0, p_1); receive(eq_1, q_1);$
$process\_fct\_c_1^1(p_1, q_1);$
$send(eq_0, p_1);$
$delay\_until(i*T_1); i := i+1;$
$receive(fq_0, p_2);$
$process\_fct\_c_1^2(p_2, q_1);$
$send(eq_0, p_2);$
$delay\_until(i*T_1);$
$while(true)\ do\ \{\ i := i+1; C_1^a\ \}$

C_1^a :
$if(L(eq_1) > 0) \quad \{C_1^b\}$
$else \quad \{C_1^c\}$
$receive(fq_0, p_2);$
$process\_fct\_c_1^2(p_2, q_{save});$
$send(eq_0, p_2);$
$q_1 := q_{save};$
$delay\_until(i*T_1);$

C_1^b :
$receive(fq_0, p_1);$
$receive(eq_1, q_2);$
$process\_fct\_c_1^1(p_1, q_2);$
$send(eq_0, p_1);$
$q_{save} := q_2;$
$send(fq_1, q_1);$
$delay\_until(i*T_1); i := i+1;$

C_1^c :
$receive(fq_0, p_1);$
$process\_fct\_c_1^1(p_1, q_1);$
$send(eq_0, p_1);$
$q_{save} := q_1;$
$delay\_until(i*T_1); i := i+1;$

We call this component *time-driven* component. Notice that we denote a $delay\_until$ action in the trace

by d. The trace set derived from the program is:
$Tr(C_1) = \{i = 1 \;\; fq_0?(p_1) \;\; eq_1?(q_1) \;\; c_1^1(p_1,q_1) \;\; eq_0!(p_1) \;\; d(i*T_1) \;\; inc(i) \;\; fq_0?(p_2)$
$c_1^2(p_2,q_1) \;\; eq_0!(p_2) \;\; d(i*T_1) \;\; (inc(i)\; t)^{\omega} | t \in Tr(C_1^a)\}$,
where $Tr(C_1^a) = \{(L(eq_1) > 0) \;\; t \;\; v | t \in Tr(C_1^b)\} \cup \{(L(eq_1) \leq 0) \;\; t \;\; v | t \in Tr(C_1^c)\}$
and $v = fq_0?(p_2) \;\; c_1^2(p_2,q_{save}) \;\; eq_0!(p_2) \;\; q_1 = q_{save} \;\; d(i*T_1)$.
$Tr(C_1^b) = \{fq_0?(p_1) \;\; eq_1?(q_2) \;\; c_1^1(p_1,q_2) \;\; eq_0!(p_1) \;\; q_{save} = q_2 \;\; fq_1!(q_1) \;\; d(i*T_1) \;\; inc(i)\}$
$Tr(C_1^c) = \{fq_0?(p_1) \;\; c_1^1(p_1,q_1) \;\; eq_0!(p_1) \;\; q_{save} := q_1 \;\; d(i*T_1) \;\; inc(i)\}$.
From the syntax of C_1, we obtain the following topology invariants:

$$0 \leq \#fq_0? - \#eq_0! \leq 1 \tag{4}$$

$$0 \leq \#eq_1? - \#fq_1! \leq 2 \tag{5}$$

5 A Characterization of Chain Execution

Our system (chain) consists of N communicating components executing in an interleaved way on the physical platform. As we have shown in [12], in order to study the behavior of this system we focus on its corresponding traceset limited to those traces that satisfy channel properties, timing properties owing to the periodic behaviour of C_1, and the priority assignment to components. We start by analyzing the trace set containing all traces corresponding to all possible arbitrary interleavings of the components actions (T_{il}), and progressively we impose invariants such that eventually we obtain the trace (and schedule) that specifies the system behavior.

We define A to be the union of the alphabets of all components (A = $\bigcup_{i=1..N} A(C_i)$), and A^{ω} the set of all infinite traces which are formed from actions in the set A. The set of traces that results from the interleaving of the components is called T_{il}:

$$T_{il} = Tr(\;||_{i=1..N}\; C_i)$$

We introduce function $\delta_p : A \times \mathbf{N} \times T_{il} \rightarrow \mathbf{N}$ that given an input stream p for each occurrence of an action from alphabet A in a trace t from T_{il}, returns the computation time needed to execute it. The computation time is expressed in CPU cycles. $\delta_p(a,k,t)$ denotes the computation time of the k^{th} occurrence of action a in trace t. Important to recognize is that the computation times of different occurrences of an action can be different. In the case of the media processing systems, the computation times of the actions are variable due to dependency on input stream contents. For the MPEG 2 streams the values of the function δ can be provided by applying techniques shown in [11].

We denote the k^{th} occurrence of an action $a_i \in A(C_i)$ in a trace t with a_i^k. From now on we take p fixed and, for ease of notation, do not write the dependence on p everywhere. Therefore for $\delta_p(a_i,k)$ we use the short hand notation $\delta(a_i^k)$. The k^{th} occurrence of a_i processes the k^{th} packet in the input stream in front of C_1([12]).

We introduce the *schedule* functions $S_{\sigma} = \{\sigma | \sigma : St(T_{il}) \rightarrow \mathbf{N}\}$ of the concurrent execution of components C_i, $i = 1..N$ on a processor. For any state s from $St(T_{il})$, a schedule function returns the finishing time of state s in a trace of T_{il}. The finishing time of a state from $St(T_{il})$ is expressed in CPU cycles, hence the domain of values of the schedule functions is the set of natural numbers.

All schedules satisfy the following *soundness* criteria:

$$\forall sa \in St(T_{il}), \sigma(sa) \geq \sigma(s) + \delta(a), \tag{6}$$

$$\sigma(sd(t)) \geq t, \forall s \in St(T_{cc}) \tag{7}$$

The meaning of (7) is that when a delay action ($d(t)$) follows a state s ($s \in St(T_{il})$) at a time earlier than t, the delay action advances the time until t. When the delay action follows s at a time equal or later than t, the delay action has no effect.

In the remainder of this document we will restrict our attention to only those schedules that indicate each action to be executed as soon possible (*eager schedules*). We focus on these schedules because they reflect the realistic execution of a streaming chain on the physical platform. Hence we define the *eager schedules* to return for every state s of a trace t the minimum of the values returned by all schedules

(in $S_{\sigma t}$) for state s:

$$\sigma_{eager}(s) = \min_{\sigma \in S_\sigma} \sigma(s)$$

The eagerness criterion imposed on the schedules in $S_{\sigma t}$ reduces the set to only one element, the eager schedule (for every state s there exists only one minimum of the values returned by the schedules in S_σ). It is also trivial to show that the eager schedule is sound as well.

We define the *response time of the chain* (RTC) for a packet k defined as the time counted from the moment that the packet starts being processed by the first action of component C_1 until the finish time of the last of C_N. RTC depends on the trace that records the chain execution, the schedule associated with this trace, and the contents of packet k which determines the computation times of each component action:

$RTC : T_{il} \times S_\sigma \times \mathbf{N} \to \mathbf{N}$, $RTC(t,\sigma,k) \stackrel{def}{=} \sigma(sfq_N!^k) - \sigma(ufq_0?^k) - \delta(fq_0?^k)$,

$sfq_N!^k, ufq_0?^k \in St(T_{il})$ and $ufq_0?^k \subseteq sfq_N!^k \subseteq t$.

In the following we restrict ourselves to the channel-consistent traces of this system, i.e. those traces in T_{il} that for all their prefixes satisfy (1) for all channels in the system. Predicate Scc specifies this constraint. The set obtained by imposing Scc on all traces from T_{il} is called T_{cc}:

$Scc(t)$: $(\forall s \in St(t), fq_i, eq_i \in A :$

$$0 \le \#(s,fq_i!) - \#(s,fq_i?) \le Cap(fq_i) \wedge$$
$$0 \le \#(s,eq_i!) - \#(s,eq_i?) \le Cap(eq_i)).$$

$T_{cc} = \{t \in T_{il} | Scc(t)\}$.

Imposing Scc limits the order in which actions can interleave, and introduces blocking. For the new constrained set T_{cc} we look again at the definitions for *ready-to-run* and *blocked* components. Given s from $St(T_{cc})$ a component C_i is *ready-to-run from channel perspective* in state s when for an action a in $A(C_i)$ such that $sa \uparrow A(C_i)$ is a state of $Tr(C_i)$ we have that $sa \in St(T_{cc})$. Also, a is called a *ready* (from channel perspective) action of C_i in state s. If sa is not an element of $St(T_{cc})$ it is not possible to execute a in the constrained set in which case we say that C_i is *blocked from channel perspective* at a in state s of T_{cc}. We say that C_i is blocked at a in state s of T_{cc}, denoted as C_i **b** a [in s of T_{cc}].

As a next step, given the schedule function σ_{eager}, we focus only on those traces in T_{cc} that satisfy the following predicate:

$S\sigma c(u) : (\forall s \in Pref(u), d(i*T_1) \in A(C_1), v \in A^\omega : u = s\,d(i*T_1)^i v : \sigma(s) \ge i*T_1 \vee Act(s) = \emptyset)$,

where $Act(s)$ is the set of actions (other than the delay action) that are *ready* from channel perspective in state s.

Predicate $S\sigma c$ imposes that a delay action $d(i*T_1)^i$ can follow a state s in a trace u, only if $\sigma(s)$ is later in time than time $i*T_1$ specified in $d(i*T_1)^i$, or there are no actions (other than delay actions) that are *ready*. $S\sigma c$ implies that a delay action can only execute at an earlier time than $i*T_1$ when there are no other actions (other than delay actions) that are ready. From here follows that in this case the time interval $i*T_1 - \sigma(s)$ is not used on the processor and we call it *idle* time. The new constraint yields another traceset $T_{\sigma c}$ that contains the *schedule consistent* traces that satisfy predicate $S\sigma c$:

$$T_{\sigma c} = \{u \in T_{cc} | S\sigma c(u)\}$$

Predicate $S\sigma c$ and the fact that every iteration of C_1 occurs every $2T_1$ implies that if an occurrence $i+1$ of an action from $A(C_1)$ follows a state $v \in St(T_{\sigma c})$ in a trace $u \in T_{\sigma c}$ then $\sigma_{eager}(v) > i*2T_1$. This is expressed as follows:

Property 1 $(\forall v \in St(T_{\sigma c}), a \in A(C_1), v\,a^{i+1} \in St(T_{\sigma c}) : \sigma_{eager}(v) > i*2T_1)$.

We are ready now to define the notion of *ready* and *blocked from time perspective* for all actions of $A(C_1)$. Any action a in $A(C_1)$ such that $s\,a^i \uparrow A(C_1)$ is a state of $Tr(C_1)$ and $s\,a^i$ is a state of $T_{\sigma c}$ (which means that $\sigma_{eager}(s) > (i-1)*T_1$) is called a *ready* (from time perspective) action of C_1 in state s. In such a situation we also say that C_1 is *ready to run from time perspective* in state s. If $s\,a^i$ is not a state of $T_{\sigma c}$ it is apparently not possible to execute a^i in the constrained set. We say that C_1 is *blocked from time perspective* at a^i in state s of $T_{\sigma c}$, denoted as 'C_1 **bt** a^i [in s of $T_{\sigma c}$]'. We also say that component C_1 is *idle* in state s.

Next we introduce an additional restriction on the traces of $T_{\sigma c}$ in the form of a priority assignment. We define priority as a function P that returns for each component a unique natural number with the interpretation that a higher number means a higher priority. This translates into the invariant that in each state of $T_{\sigma c}$ where there are multiple components ready to run, an action of the one with the highest priority is selected. Obviously the action selected cannot be a delay action (as long as there do exist other ready regular actions) because all traces in $T_{\sigma c}$ satisfy predicate $S\sigma c$. The imposed invariant is then specified as follows:

$$Scp(t) : (\forall s \in Pref(t), a_i u \in A^{\omega} : t = sa_i u \wedge a_i \in A(C_i) : P(Comp(a_i)) = \max_{C \in RR(s)} P(C)).$$

Limiting $T_{\sigma c}$ according to Scp gives T_{pc}, the *priority consistent* traces: $T_{pc} = \{\forall t \in T_{\sigma c} | Scp(t)\}$.

Property 2 *T_{pc} has precisely one element.*

Proof Identical with the proof presented in [12]. □

We denote the unique trace in T_{pc} with ρ.

5.1 Characterization of the unique trace ρ

5.1.1 QoS requirements

In the case of the chain where the first component is time driven the QoS requirement must demand that C_1 executes periodically, at a strict rate T_1. To explain this requirement we consider the case of a video processing chain, where C_1 is a video digitizer. If C_1 does not execute at a strict rate T_1, it will not be able to capture video frames at the correct rate, which induces video artefacts such as breaks in the movement of the video objects in the image, or in more severe situations, abrupt changes of image when the captured video contents is eventually displayed on a screen. Obviously all these artefacts imply a lower perceived QoS from the perspective of the human user.

The QoS requirement is expressed as:

$$SQoSc(t) : (\neg((L(eq_1) \leq 0)\ in\ t) \wedge (\forall_{u_1,u_2 \in Pref(t)} \wedge\ u_1 = v_1\ fq_0?(p_1) \wedge u_2 = v_2\ fq_0?(p_2) : \sigma(u_1) - \sigma(u_2) = T_1))), t \in Pref(\rho).$$

The meaning of the predicate above implies that if $SQoSc(\rho)$ holds, then action $(L(eq_1) \leq 0)$ does not occur in ρ (program C_1^c is never executed during ρ), and C_1 executes $fq_0?$ always at time interval T_1.

We denote with S_i^k the sum of the computation times of actions during the k^{th} loop iteration of component C_i, $\quad 1 \leq i \leq N$. We also define S as $\sum_{i=1}^{N} S_i$.

We consider for the beginning fixed computation times for all component actions. In the remainder of this section we show that $S \leq T_1$ is a sufficient condition such that the *QoS* requirement would hold for the entire ρ. To prove this, we analyze first the behaviour of the chain when $S \leq T_1$ is imposed. We denote with *RSC* the sub-chain composed of components $C_2, ..., C_N$.

Theorem 1 *When $S \leq T_1$ and $Cap(eq_1) \geq 2$, the pipeline system in which the first component has a periodic behavior, assumes a repetitive, periodical behavior called stable phase after a finite initial phase. The complete behavior is characterized by*

$\rho = t_{init}\ (t \quad t_R \quad d(i*T_1) \quad v \quad d(i*T_1))^{\omega}$ *where*
$t = (inc(i) \quad L(eq_1) > 0 \quad fq_0?(p_1) \quad eq_1?(q_2) \quad c_1^1(p_1,q_2) \quad eq_0!(p_1) \quad q_{save} = q_2 \quad fq_1!(q_1))$,
$v = (inc(i)\ fq_0?(p_2) \quad c_1^2(p_2,q_{save}) \quad eq_0!(p_2) \quad q_1 = q_{save})$,
$t_{init} = ((i = 1) \quad fq_0?(p_1) \quad eq_1?(q_1) \quad c_1^1(p_1,q_1) \quad eq_0!(p_1) \quad d(i*T_1) \quad inc(i) \quad fq_0?(p_2)$,
$\quad c_1^2(p_2,q_1) \quad eq_0!(p_2) \quad d(i*T_1))$,
and t_R is the trace recording the interleaved execution of one loop iteration of components in sub-chain RSC.

Proof We denote with t^i and v^i the sub-traces composed of the i^{th} occurrences of their corresponding actions.

The assumption about the initial state of the queues in the chain (excepting fq_0 and eq_0) is that all forward queues are empty and all backward queues are full. This means that the initial state of all components except C_1 is that they are blocked at action $fq_{i-1}?$ (C_i **b** $fq_i?, 2 \leq i \leq N$).

Trace ρ begins with t_{init} (Figure 3), which consists of the first actions of component C_1 occurring during the first $2T_1$ according to the program of C_1. During this $2T_1$ time no actions of components $C_i, 2 \leq i \leq N$ occur because all components are still blocked (no full packet has been produced in fq_1 yet). In fact, up until the first occurrence of $fq_1!(q_1)$, only actions of C_1 can follow t_{init} because all the other components are blocked. Note that at the end of t_{init}, $L(eq_1) > 0$ ($Cap(eq_1) \geq 2$) which implies that t^1 is the sub-trace following t_{init} and ending with $fq_1!(q_1)^1$ according to the program of C_1.

Producing a full packet in fq_1 causes a de-bloking in cascade of components $C_i, 2 \leq i \leq N$. All these components can execute one iteration of their loop (which takes less than S time). The subtrace recording the interleaved execution of the first iteration of components $C_i, 2 \leq i \leq N$ is t_R. The order of execution of components in t_R is entirely determined by the priority assignment.

Note that $\sigma(t_{init} \quad t^1 \quad t_R) < \sigma(t_{init} \quad t^1) + S$ and $S < T_1$ imply that $\sigma(t_{init} \quad t^1 \quad t_R) < \sigma(u \; fq_1!(q_1)^2)$, ($u \in St(\rho)$) because we know that $fq_1!(q_1)$ occurs only at most once every $2T_1$.

The last inequality implies that t_R ends before $fq_1!(q_1)^2$ can be executed, which means that by the time all components in *RSC* have executed one loop iteration no other full packet has been produced in fq_1. This means that after executing one loop iteration, components $C_i, 2 \leq i \leq N$ become blocked at action $fq_{i-1}?$ again.

In the time interval $[\sigma(t_{init} \quad t^1 \quad t_R), \; 3T_1)$ (as an effect of action $d(3T_1)$) the system experiences an interval of idle time. At the end of this interval according to its program C_1 executes the sub-trace v^1.

At the end of v^1, as an effect of action $d(4T_1)$, the system experiences again idle time until time $4T_1$(all data-driven components are blocked). Note that between time $2T_1$ and $4T_1$ (when t_R is executed), C_1 recycles 1 empty packet in eq_1, which means that at time $4T_1$, $L(eq_1) > 0$. This means that at time $4T_1$ component $C1$ executes sub-trace t^2, at the end of which it produces again 1 full packet in fq_1. The production of a full packet in fq_1 causes again a de-blocking in cascade of components in *RSC* during which another loop iteration of all components is executed. The execution of t_R is again followed by idle time, v^2, and again idle time for the same reasons explained above. The repetition of this execution lasts as long as there is input in fq_0. Because of our assumption that fq_0 always contains input, the suffix ($t \quad t_R \quad d(i*T_1) \quad v \quad d(i*T_1)$) is repeated infinitely. We denote the trace recording the repetitive execution with s.

Note that $S \leq T_1$ implies that C_1 is *ready to run* from time perspective at least every $2T_1$. We have seen that C_1 can never become blocked from channel perspective, which means that the execution recorded by s is not only repetitive, but also periodic. □

The following corollary states that when $S \leq T_1$, whenever C_1 executes $fq_0?(p_1)$, all the other components in the chain ($C_i, 2 \leq i \leq N$) are blocked at action $fq_{i-1}?$.

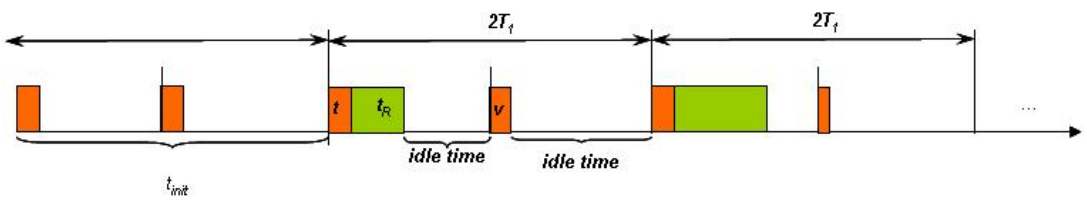

Figure 3. Execution of trace ρ

Corollary 1 $S \leq T_1 \wedge L(fq_1) \geq 2 \wedge t \in Pref(\rho), t = s \;\; fq_0(p_1)? \Rightarrow C_i \; \boldsymbol{b} \; fq_{i-1}?, \forall i, 2 \leq i \leq N.$

Proof Follows directly from the discussion presented in the proof of Theorem 1 □

Corollary 2 $S \leq T_1 \wedge P(C_1) = \max_{i=1}^{N} P(C_i) \Rightarrow SQoSc(\rho)$

Proof We have seen from Theorem 1 that action $(L(eq_1) \leq 0)$ is not in ρ.
Additionally, $S \leq T_1$ and $P(C_1) = \max_{i=1}^{N} P(C_i)$ imply that C_1 is not only ready to run from time perspective and channel perspective every T_1, but will also execute at strict rate T_1. This means that $SQoSc(\rho)$ holds. □

When considering variable computation times for the actions of components by imposing $S^M \leq T_1$ where $S^M = \sum_{i=1}^{N} S_i^M$, $S_i^M = \max_{k \in N} S_i^k$, the stable phase is reached again and all lemmas and corollaries presented above hold as well, which means that again the trace ρ satisfies the *QoS* requirement. Indeed by ensuring at design time that $S^M \leq T_1$ where here S^M takes into account the worst case computation time for all actions of components, the reasoning and the conclusions about the execution of the chain presented above hold again. Such a restriction in the design of a chain can be verified by measuring the load on the processor of one iteration of each component.

5.1.2 Practical Applications

Trace ρ and its eager schedule can be calculated when knowing the computations times of all component actions, and by choosing in each state the ready action of the component with the highest priority. Knowing the trace and the schedule allows to calculate the number of context switches, and the number of actions needed to process a packet from input to the output of a chain. The schedule also renders the start and response times for individual tasks and the response time of the chain.
Important to note is that because $S^M \leq T_1$, regardless of its priority, component C_1 has the same effect on the interleaved execution of the rest of components during the stable phase as a data-driven C_1 with minimum priority (as described in [12]). Owing to this fact, we find that corollaries addressing the stable phase of a chain composed of only data-driven components (with C_1 having the lowest priority) hold in this case as well. From here we deduce that:

- the minimum necessary and sufficient capacity of each queue in the chain except fq_1 and eq_1 is 1 (Corollary 9, [12]).
- the minimum necessary and sufficient capacity of queues fq_1 and eq_1 is 2 (follows directly from topology invariant expressed in (5)).
- the response time of the chain is to be calculated as shown in [12], in the case of a chain composed of only data driven components with $m = 1$. Also follows directly that the response time of the chain cannot be improved more because assigning the minimum priority to C_1 as suggested in Theorem 13 [12] is not necessary anymore.
- the number of context switches occurring due to the interleaved execution of the data driven components during the stable phase cannot be improved more because assigning the minimum priority to C_1 as suggested in Theorem 12, i. [12] is not necessary anymore.

6 Conclusions

We have presented a model for the dynamic behavior of media processing chains of software components communicating via bounded buffers. The first component is time-driven (has a periodic behavior) and the rest of components are data driven. Components can have variable computation times.
We specify the behavior of the chain by means of a trace ρ of the actions of the components that make up the chain, and the associated schedule function. We have proven that when overload situations are prevented at chain design time, after a finite prefix (initial phase) the trace recording the execution of

the chain becomes repetitive and periodic(stable phase). We have also shown what are the conditions in which the chain satisfies the quality of service requirements for the entire trace ρ. In addition, we have explained that the time-driven component has the same effect on the interleaved execution of the rest of components during the stable phase as a data-driven C_1 with minimum priority (as described in [12]). This reduces the analysis of this time-driven system to be identical to that of the data-driven system in [12]. As a result the issues related to the calculation and optimization of the necessary and sufficient memory in each buffer, the response time of the chain and the number of context switches occurring during the chain execution are to be reasoned about in the same way.

Note that the $S < T_1$ condition is rather strict. A further step is to investigate how to satisfy QoS requirements when $S < 2T_1$. Our investigations until now reveal that in such a case trace $\rho = t_{init}\ (t \quad t_{R1} \quad d(i*T_1) \quad v \quad t_{R2} \quad d(i*T_1))^{\omega}$. The reason for the two sub-traces t_{R1} and t_{R2} recording the cascade execution of components in *RSC* is that C_1 can preempt one of these components when it becomes ready to run from time perspective. Note that in such a case simply assigning the highest priority to C_1 in order to satisfy the QoS requirement is not enough. That is because of the atomicity of actions of data-driven components. Because of this fact, when C_1 becomes ready to run from time perspective it cannot preempt exactly at time $i*T_1$. This means that a C_1 execution at a strict rate cannot be accomplished. One solution is to renounce the atomicity of actions for the data-driven components and another is to relax the QoS requirements to allow a small deviation from the strict rate. All this analysis is subject for further work.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd., 1996.

[2] G. v. Doren and B. Engel. *Streaming in Consumer electronic Products Beyound Processing Data. In Dynamic and Robust Streaming in and Between Connected Consumer Electronic Devices*. Editor P.D.V. v.d. Stok, Philips Research Series, Springer, 2005.

[3] R. Gerber and S. Hong. Semantic-based compiler transformations for enhanced schedulability. In *Proceedings of IEEE Real-Time Systems Symposium 1993*, pages 232–242, 1993.

[4] S. Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 60–71, June 1997.

[5] M. Gonzalez Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings of the IEEE Real time Systems Symposium*, pages 116–128, December 1991.

[6] A. M. Groba, A. Alonso, J. A. Rodriguez, and M. Garcia-Valls. Response time of streaming chains: Analysis and results. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 182–192, 2002.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[8] M. K. Klein, T. H. Ralya, B. Pollac, R. Obenza, and M. Gonzalez-Harbour. *A practitioner's Hanbook for Real-Time Analysis: Guide to Rate monotonic Analysus for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[9] C. Mercer, S. Savage, and H. Tokuda. Processor capability reserves: Operating system support for multimedia applications. In *International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[10] C. Otero-Perez, E. Steffens, P. v. Stok, S. Loo, A. Alonso, J. Ruiz, R. Bril, and M. Garcia Valls. *QoS-based Resource Management for Ambient Intelligence, Ambient Intelligence: Impact on Embedded System Design*. Kluwer Academic Publishers, pages 159-182, 2003.

[11] S. Peng. Mpeg2 decoding with graceful degradation. In *Proceedings of Philips Digital Video Technologies 2000 Workshop*, 2000.

[12] M. A. Weffers-Albu, J. J. Lukkien, E. F. M. Steffens, and P. D. V. v. Stok. On a theory of media processing systems behavior, with applications. In *Proceedings of the Euromicro Conference on Real Time Systems*, 2006, in press.