On a Theory of Media Processing Systems Behavior, with Applications

M.A. Weffers-Albu[1], J.J. Lukkien[1], E.F.M. Steffens[2], P.D.V. van der Stok[1,2]
[1]Technische Universiteit Eindhoven, [2]Philips Research Laboratories
{m.a.albu, j.j.lukkien}@tue.nl,{liesbeth.steffens, peter.van.der.stok}@philips.com

Abstract

In this article we provide a model for the dynamic behavior of media processing chains of tasks communicating via bounded buffers. 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. From this we derive corollaries that give design guidelines, supporting the goal of improving chain end-to-end response time and the optimization of resources, which is achieved by minimizing context-switching overhead and buffer sizes.

1. Introduction

1.1. Context

For the media processing in its Nexperia platforms, Philips has adopted a *Pipes and filters* architecture[1], known as TSSA. With this architecture, a media-processing system can be viewed as a graph in which nodes represent software components and edges represent buffers. Each component corresponds to a task, and the communication between tasks is buffered. Although hardware accelerators perform part of the media processing, the actual streaming is done in software on a single processor. To enable the construction of many complex systems in short time, Philips Research is investigating the composition of systems from reusable subsystems that share the platform resources. The resource sharing is based on the *resource reservation* paradigm [2] at the subsystem level [3]. Each subsystem represents a temporally independent sub-graph (the most common being a linear chain of components), and is provided with a *guaranteed resource budget*. Within each subsystem, the tasks are scheduled using *fixed priority scheduling*. Ideally, the behavior of each subsystem is dynamically controlled to ensure that the subsystem stays within its timing constraints ([4], [5]).

Due to the dependencies between the tasks, and their different behaviors, it is difficult to predict the behavior of the subsystem. Hence, it is difficult to determine resource budgets, to predict response times, to minimize buffer sizes and context switch overhead, and to reason about subsystem composition. The current practice lacks a theoretical underpinning that helps designers and integrators beyond intuition and experience, and is yet easy to understand. Such a theory is also needed to control the behavior at run time, to make sure timing requirements are met even in overload conditions.

1.2 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 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 graphs 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 graphs that include branching and more complex types of components. This paper presents the initial case we analyzed, of a linear chain executing in a cooperative environment. Although this case misses a number of issues typical for media processing systems (such as variable computation times of tasks and timing constraints of the chain), it is essential for laying the foundation of our theory, for understanding what causes the repetitive behavior, and how this behavior can be controlled.

Our approach allows us to calculate the execution order of the components in a chain, expressed as a trace of actions [13] taken by each component. We formally prove that the behavior of the chain can be expressed as a unique trace, which assumes a repetitive pattern after a finite prefix. The trace is completely determined by the individual traces of the components, the topology of the chain, the capacities of the communication buffers, and the static priorities of the components. The unique trace of actions proves an excellent starting point for

further analysis. Simple additive formulas for the start times and response times of the individual tasks 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 (due to the buffered communication), we calculate the necessary and sufficient capacities for each buffer in the chain.

Our choice to formally express the behavior of the system in terms of traces comes from the fact that traces provide an adequate and simple way to specify the behavior of our streaming chains. Indeed other formal techniques such as Petri Nets [6] lead to very complex diagrams even when describing chains composed of quite simple components. CSP [13] presents another complication in the fact that buffers are considered processes (tasks) as well. In this case the tasks corresponding to components are assigned priorities while those corresponding to buffers do not. The complication comes from the fact that it is difficult to analyze the behavior of a system in which some tasks have priorities and others not.

This paper is organized as follows. In section 2 we discuss related work. Section 3 gives insight about basic notions of trace theory and sets the general approach for our work. Section 4 presents the system architecture we use and initial properties of the chain behaviors. Section 5 provides a characterization of the unique trace that expresses the chain behavior and answers to practical questions regarding the calculation and optimization of queue capacities, the length of the initial phase and the number of context switches. In section 6 we present timing aspects of the chain behavior and we show how the response time of the chain can be optimized. We end our presentation with conclusions in section 7.

2. Related work

Several attempts have been made to analyze message passing, streaming systems. Closely related work [7] considers also an execution model for video streaming chains inspired by TSSA. The article [7] 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 [8].

Klein et al. [10] 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 [11] 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 [10]. 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 provided the theory that helps engineers to reason rigorously 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 [12] 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

3.1. Components and traces

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 $\mathcal{A}(C)$. Alphabets of different components are disjoint. The semantics is 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 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 $Tr(C):\{(ab)^{\omega}\}$.

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

{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.

3.2. 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}*.

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

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 [13]. However, in contrast to CSP we use bounded channels by imposing the invariant

(1) $\quad 0 \leq \#c! - \#c? \leq Cap(c),$

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 the result of a parallel composition C_0 || C_1. Any action a in $\mathcal{A}(C_0)$ such that $sa\uparrow\mathcal{A}(C_0)$ 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 **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. Hence, for an action $a_i \in A(C_i)$, $Comp(a_i) = C_i$.

Furthermore, consider a trace t written as $t = s_0\ a\ b\ s_1$. 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 finite trace from a traceset T, and returning the *number of context switches* occurring in the trace:

$NCS : T \rightarrow \mathbb{N}$, $NCS(\varepsilon) = 0$, $NCS(a) = 0$,

$$NCS(abt) = \begin{cases} NCS(bt) & \text{if } Comp(a) = Comp(b) \\ NCS(bt) + 1 & \text{otherwise.} \end{cases}$$

3.3. Approach

The model presented above is a fairly conventional interleaving model [13]. Compared to regular trace semantics there are two major differences. First, we use the complete executions according to the syntax as the semantics, rather than the prefix closure. Secondly, we do not introduce any synchronization concepts in the syntax or the trace semantics. Instead we introduce the interpretation of atomic actions as well as execution policies by limiting attention to those traces that satisfy the interpretation. This makes the semantics simpler and the manipulation easier.

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 priority assignment;
- the timing assignment.

4. A Streaming Pipeline

4.1. System architecture

We focus on systems consisting of a collection of communicating components connected in a pipelined fashion (*Pipes and Filters* architecture style). An instance of this architecture style, the *TriMedia Streaming Software Architecture* (TSSA) 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.

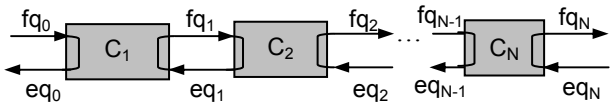

Figure 1. Chain of components.

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. The system executes in a cooperative environment meaning that the environment will always provide input and always accept output. That means that blocking on the queues fq_0, eq_0, fq_N, eq_N, is not possible. 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 in the following property:

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

The behavior of C_i, $1 \leq i \leq N$, 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.

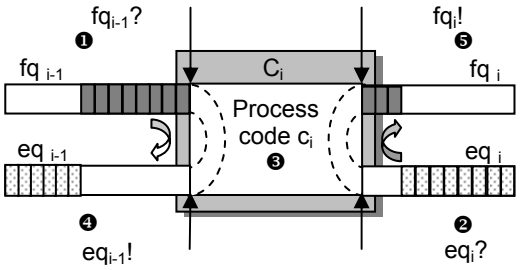

Figure 2. A basic streaming component.

The syntax for the execution of a component C_i and the corresponding 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}, 1);send(fq_i, 1); },

with traceset $Tr(C_i)$: $\{ (fq_{i-1}?, eq_i?, c_i, eq_{i-1}!, fq_i!)^{\omega}\}$

for $1 \le i \le N$. Note that we write the atomic actions differently in the program syntax than in the trace in order to make the distinction clear.

We denote $(fq_{i-1}?\ eq_i?\ c_i\ eq_{i-1}!\ fq_i!)^{\omega}$ with t_i ($1 \le i \le N$). All t_i traces are repetitive and the sequence of actions that represent one iteration is $fq_{i-1}?, eq_i?, c_i, eq_{i-1}!, fq_i!$.

From the syntax of C_i we obtain the following topology invariants.

(2) $0 \le \#fq_{i-1}? - \#eq_{i-1}! \le 1$

(3) $0 \le \#eq_i? - \#fq_i! \le 1$

For convenience, we give names to these differences and call them x_i and y_i respectively. The four different values of this pair correspond to different states or positions within one iteration of trace t_i.

4.2. Channel constraints

We define A to be the union of the alphabets of all components ($A = \bigcup_{i=1..N} \mathcal{A}(C_i)$), 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}:

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

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}:

$Scc(t)$: $(\forall s \in Pref(t)$: $0 \le \#(s,fq_i!) - \#(s,fq_i?) \le Cap(fq_i)$
$\wedge\ 0 \le \#(s,eq_i!) - \#(s,eq_i?) \le Cap(eq_i))$.

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

Imposing *Scc* limits the order in which actions can interleave, and introduces blocking. For the new constrained set T_{cc} we recapitulate the definitions for *ready-to-run* and *blocked* components. Given s from $St(T_{cc})$, a component C_i is *ready-to-run* in state s when for an action a in $\mathcal{A}(C_i)$ such that $sa \uparrow \mathcal{A}(C_i)$ is a state of $Tr(C_i)$ we have that $sa \in St(T_{cc})$. Also, a is called a *ready* 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* at a in state s of T_{cc}.

Property 1. For component C_i ($1 \le i \le N$), blocking at $eq_{i-1}!$ is not possible.

Proof. We prove this by contraposition by considering a state s where C_i would be blocked at $eq_{i-1}!$ and showing this state cannot exist. To be blocked at $eq_{i-1}!$, execution of C_i has to proceed until just before $eq_{i-1}!$. This means that for this state s we have

(4) $x_i = 1$

In addition, blocking is caused by the channel consistency, hence

(5) $\#eq_{i-1}! - \#eq_{i-1}? = Cap(eq_{i-1})$

We compare the situation between C_i and C_{i-1}

0
$< \{ (4), (3) \}$
$x_i + y_{i-1}$
$= \{ (2), (3) \}$
$\#fq_{i-1}? - \#eq_{i-1}! + \#eq_{i-1}? - \#fq_{i-1}!$
$= \{ (5) \}$
$\#fq_{i-1}? - \#fq_{i-1}! - Cap(eq_{i-1})$

The last term is at most 0 according to (1), which is a contradiction. □

By symmetric reasoning we also have:

Property 2. For component C_i, blocking at $fq_i!$ is not possible.

From these two properties we conclude that when blocking of components due to communication occurs, it is possible only at input actions.

4.3. Precedence order constraints

Next we introduce an additional restriction on the traces of T_{cc} 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_{cc} where there are multiple components ready, an action of the one with the highest priority is selected. The imposed invariant is then specified as follows:

$Scp(t)$: $(\forall s \in Pref(t)\ , a_i \in \mathcal{A}(C_i)$: $t = sa_iu$:

$$(Comp(a_i)) = \max_{C \in RR(s)} P(C)).$$

Limiting T_{cc} according to *Scp* gives T_{pc}, the *priority consistent* traces:

$T_{pc} = \{t \in T_{cc} | Scp(t)\}$.

Property 3. T_{pc} has precisely one element.

Proof. We wish to prove that $| T_{pc} | = 1$.

In the first part of the proof we show that T_{pc} consists of *at least* one element by construction according to the definition of *Scp,* following the algorithm below:

$\rho_0 := \varepsilon;$

$$\rho_{n+1} := \begin{cases} \rho_n\, a, & \text{if } \exists a: P(Comp(a)) = \max_{C \in RR(\rho_n)} P(C) \\ \rho_n, & \text{otherwise} \end{cases}$$

We consider $\rho = \lim_{n \to \infty} \rho_n$.

$Scp(\rho_i)$ holds for all i, and each ρ_i is a prefix of ρ, hence $Scp(\rho)$ holds as well. Therefore $| T_{pc} | \geq 1$.

In the second part of the proof we show by contraposition that $| T_{pc} | \leq 1$.
We assume:

$(\exists\, t_1, t_2 \in T_{pc}, t_1 \neq t_2$:
$(\forall s, a_i, u: t_1 = s a_i u \wedge a_i \in \mathcal{A}(C_i))$:
$P(Comp(a_i)) = \max_{C \in RR(s)} P(C)) \wedge$
$(\forall s, a_l, w: t_2 = s a_l w \wedge a_l \in \mathcal{A}(C_l)$:
$P(Comp(a_l)) = \max_{C \in RR(s)} P(C)))$

Consider the first state s where the next action of the two traces differ, resp. a_i and a_l. The fact that the priorities assigned to all components are unique and given that $P(Comp(a_i)) = P(Comp(a_l))$ implies that $Comp(a_i) = Comp(a_l)$. This implies that in state s there do exist two *ready* actions of the same component, which is impossible according to the definition of *ready* action. □

We denote the unique trace in T_{pc} with ρ and we are interested in the occurrence of context switches in ρ. Consider a context switch between components C_i and C_j, i.e., $\rho = s a_i b_j u$ with $s a_i b_j \in Pref(\rho)$, $u \in A^{\omega}$, $a_i \in \mathcal{A}(C_i)$ and $b_j \in \mathcal{A}(C_j)$.

If $P(C_i) < P(C_j)$ we say that C_j *preempts* C_i, we call this situation *preemption* and the context switch occurs *due to preemption.* In the other case we call this a *context switch due to blocking.*

5. Characterization of the unique trace ρ

The next goal is to characterize the single trace ρ as a function of the priority assignment. In the first sub-section we present two lemmas and a theorem that allow the calculation of the trace at design time. In the second sub-section we show how the theorem helps answer practical questions about the minimum sufficient amount of memory, calculating and optimizing the length of the initial phase in the trace and minimizing the number of context switches.

5.1. Stable phase characterization

Consider a component C_i that is a left local minimum in terms of priority $(\forall j: j<i: P(C_j) > P(C_i))$. Whenever it is executing the components to the left of it are blocked. However, the blocking can only be at a single place, according to the following lemma.

Lemma 4. Let C_i be such that $(\forall\, j: j<i: P(C_j) > P(C_i))$ and consider a state s in $St(\rho)$ such that the next action after s in ρ is one of $\mathcal{A}(C_i)$. Then C_j ***b*** $eq_j?$ [in s of T_{cc}] for all $j<i$.

In words, the lemma states that whenever actions of this local minimum component are executed, the components to the left of it are blocked at reading an empty packet from their right neighbors. The lemma specifies C_j ***b*** $eq_j?$ [in s of T_{cc}] in order to clarify the blocking context given by T_{cc}. Indeed the blocking is due to the fact that predicate $Scc(t)$ where $t = s\ eq_j?$ does not hold.

Proof. We prove this by contraposition again. Let C_j be the leftmost component that is blocked elsewhere. Properties 1 and 2 already showed that blocking at output is not possible. The only possible actions where blocking due to channel communication is possible is either at action $fq_{j-1}?$ or $eq_j?$. For C_1, blocking at $fq_0?$ is not possible because of the cooperative environment hence, $eq_1?$ is the only possible place to block. Therefore we must assume, $j>1$ and C_j ***b*** $fq_{j-1}?$. This implies that $y_{j-1} = 0$, $x_j = 0$, $\#fq_{j-1}! - \#fq_{j-1}? = 0$, and $\#eq_{j-1}! - \#eq_{j-1}? = Cap(eq_{j-1})$.

From this follows:

$$\begin{aligned} & 0 \\ = & \\ & x_j + y_{j-1} \\ = & \{ (2), (3) \} \\ & \#fq_{j-1}? - \#eq_{j-1}! + \#eq_{j-1}? - \#fq_{j-1}! \\ = & \{ \#fq_{j-1}! - \#fq_{j-1}? = 0, \text{ and} \\ & \#eq_{j-1}! - \#eq_{j-1}? = Cap(eq_{j-1}) \} \\ = & \\ & -Cap(eq_{j-1}). \end{aligned}$$

This is in contradiction with $Cap(eq_{j-1})$ being strictly positive. □

We have a symmetric lemma for C_i as local right-minimum in a chain $(\forall j: j>i: P(C_j) > P(C_i))$.

Lemma 5. Let C_i be such that $(\forall\, j: j>i: P(C_j) > P(C_i))$ and consider a state s in $St(\rho)$ such that the next action after s in ρ is one of $\mathcal{A}(C_i)$. Then C_j ***b*** $fq_{j-1}?$ [in s of T_{cc}] for all $j>i$.

Proof. Analogous to proof of *Lemma 4*.

Lemma 4 and *5* form the key for understanding the behavior of the whole pipeline. Consider component C_m with minimal priority in the entire chain

($P(C_m) = \min_{i=1..N} P(C_i)$). The behavior of C_m is given by $(fq_{m-1}?, eq_m?, c_m, eq_{m-1}!, fq_m!)^\omega$; *Lemma 4* gives us that before the first execution of $fq_{m-1}?$, all components C_j to the left of C_m have becomes blocked at $eq_j?$ and all components to the right of C_m are blocked at $fq_{j-1}?$. The unique trace recording the interleaved execution of components up to the first execution of an action of C_m is composed of $5 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} Cap(eq_i) + (m-1)$ actions (see *Property 10* below), executed in an order determined by the priority of the components left of C_m. We denote this trace by t_{init} and we call it the *initial phase* in the execution of the system.

After t_{init}, a few actions of C_m occur until $eq_{m-1}!$ Immediately after that component C_{m-1} runs; actually, all components C_j to the left of C_m will execute an iteration and return to being blocked at $eq_j?$. The order of these actions is again entirely determined by the priority assignment and is of length $5(m-1)$. We call this trace t_L and we denote the set of components to the left of C_m with S_L. Next, C_m outputs: $fq_m!$, which results in the enabling of the components to the right of C_m that execute a full iteration until each component is blocked at the same place again. This trace has length $5(N-m)$ and we call it t_R (we denote the set of components to the right of C_m with S_R). At this point C_m is again the only component *ready-to-run* and the situation described above repeats. The order of actions in the traces t_L and t_R is always the same because the priority assignment does not allow any deviation. In short this means that the system follows a repetitive behavior. We arrive at the following theorem.

Theorem 6 (Stable Phase Theorem). The pipeline system assumes a repetitive behavior after a finite initial phase. The complete behavior is characterized by

$$\rho = t_{init}\ (fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ t_L\ fq_m!\ t_R)^\omega$$

We denote the trace $(fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ t_L\ fq_m!\ t_R)^\omega$ with t_{stable} and we call it the *stable phase* in the execution of the system.

Corollary 7. The stable phase starts when C_m executes for the first time.

5.2. Practical applications

5.2.1. Channel usage, minimizing channel capacities

The following corollary shows what the number of packets is in all queues at the beginning of any iteration of t_{stable}, more precisely right after C_m executes $fq_{m-1}?$. This information becomes useful later when we address issues related to the chain response time for each packet in the chain (Section 6).

Corollary 8. Given a state s in $St(\rho)$ such that $s = t_{init}\ (fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ t_L\ fq_m!\ t_R)^j\ fq_{m-1}?$, the following statements hold:

a. $\forall i: 1 \le i < m: L(fq_i) = Cap(fq_i) - 1 \wedge L(eq_i) = 0$

b. $\forall i: m \le i < N: L(fq_i) = 0 \wedge L(eq_i) = Cap(eq_i)$.

Proof.

a. When C_m executes $fq_{m-1}?$, components C_i ($1 \le i < m$) are blocked at $eq_i?$ *(Lemma 4).* That implies that $L(eq_i)=0$. From here follows that there are $Cap(fq_i)$ full packets available for consumption. Given the order of actions in the individual traces of components, each C_i executes first $fq_{i-1}?$ before $eq_i?$, meaning that when C_i becomes blocked at $eq_i?$ it will have already executed $fq_{i-1}?$, which makes that each fq_i contains $Cap(fq_i)-1$ full packets and one packet is inside the component C_i.

b. Lemma 5 implies that when C_m executes $fq_{m-1}?$ components C_i ($m < i \le N$) are blocked at action $fq_{i-1}?$. This implies that $L(fq_{i-1}) = 0$ and $L(eq_{i-1}) = Cap(eq_{i-1})$. Note that no empty packet is inside C_i when it becomes blocked at $fq_{i-1}?$ because again, given the order of actions in the individual traces of components, each C_i executes first $fq_{i-1}?$ before $eq_i?$. □

Corollary 9 shows the minimum necessary and sufficient capacity of the queues that interconnect the components. It is relevant to know the minimum amount of memory that the system needs for communication between components.

Corollary 9. The minimum queue capacity necessary and sufficient for any queue fq_i (and implicitly eq_i) $1 \le i < N$, is 1.

Proof. Follows directly from *Corollary 8.*

5.2.2. Optimizing the initial phase length

In this subsection we calculate the length of the initial phase (*Property 10*) and show how t_{init} can be reduced and even eliminated (*Corollary 11*). Reducing the initial phase is important because designers assign budgets to the chain based on the resource requirements of the stable phase.

Property 10. The length of the initial phase in number of actions is

$$5 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} Cap(eq_i) + (m-1).$$

Proof. Due to lack of space we give a sketched proof below, the full proof is available on demand. The number of actions executed until the execution of the system reaches the stable phase is the number of actions executed by all C_i ($1 \le i < m$) until all their backward input queues (eq_i, $1 \le i < m$) are drained. All these queues are drained after $5 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} Cap(eq_i)$ actions.

After that, all *m-1* components C_i ($1 \le i < m$) will execute action $fq_{i-1}?$ and then they become blocked at action $eq_i?$ ($1 \le i < m$). That leaves us with the total number of actions for the initial phase:

$$|t_{init}| = \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} Cap(eq_i) + (m-1). \quad \square$$

Corollary 11. $|t_{init}|$ can be reduced by decreasing m or decreasing the capacity of the queues to the left of C_m in the chain.

Proof. Direct consequence of *Property 10*. $\square$

5.2.3. Minimizing the number of context switches

In this subsection we show that the priority assignment influences the number of context switches in ρ and we give insight on what is the appropriate priority assignment that minimizes $NCS(t_{init})$ and $NCS(t_{stable})$.

Theorem 12.

a. Minimum $NCS(t_{init})$ is achieved when $P(C_1) = \min_{i=1..N} P(C_i)$.

b. The minimum NCS during one iteration of t_{stable} can be achieved either when:

i. $P(C_1) = \min_{i=1..N} P(C_i)$.

or when

ii. $P(C_N) < P(C_1) < P(C_2) < \ldots < P(C_{N-1})$ with $\forall i: 1 \le i < N-1,\ Cap(fq_i) = 2$.

Proof.

a. *Corollary 11* implies that for $m=1$, the initial phase is eliminated, therefore $NCS(t_{init})$ is 0.

b. *Theorem 6* shows that regardless of the priority assignment, context switches occurring during the execution of components C_i ($i \neq m$) due to blocking cannot be avoided; hence the only context switches that can be eliminated are the ones due to preemption. In the case of C_m, this component does not block but is preempted after both actions $eq_{m-1}!$ and $fq_m!$. Only one of these two context switches can be avoided by choosing $m = 1$ or $m = N$.

In case *i.*, $m = 1$. In this case, regardless of the priority assignment to the components C_i ($i > m$), the plus of context switches due to preemption is already eliminated because preemption and blocking occur after the same action ($fq_i!$), therefore $NCS(\rho)$ cannot be optimized more.

In case *ii.* , $m = N$. In this case, preemption and blocking occur after different actions. The priority assignment suggested implies that all context switches caused by preemptions are avoided in trace t_L, each component executes until it is again blocked. Moreover, with this priority assignment, had the lengths of the queues been 1, additional context switches would have occurred due to the blocking on the input forward queues of components $C_2,\ldots,C_{N-1}$ (*Corollary 8*) therefore the additional condition on the minimal length of queues must be imposed for case *ii:* $\forall i: 1 \le i < N-1, Cap(fq_i)=2$. $\square$

Tradeoff

a. The priority assignment suggested by *Theorem 12 - ii* implies that all context switches caused by preemptions were eliminated, at the cost of an initial phase.

b. The priority assignment suggested by *Theorem 12 - ii.* achieves a minimum $NCS(t_{stable})$ at the cost of more memory needed ($\forall i: 1 \le i < N-1, Cap(fq_i)=2$).

6. Timing

In order to analyze the system from the perspective of time we extend our modeling to include time. Using this we explain how the chain response time can be optimized.

6.1 Theoretical concepts

We introduce function $\delta : A \times \mathbb{N} \times T_{il} \rightarrow \mathbb{N}$ that 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. $\delta(a, k, t)$ 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. For the MPEG 2 streams the values of function δ can be provided by applying techniques shown in [14].

We denote the k^{th} occurrence of an action $a_i \in \mathcal{A}(C_i)$ in a trace t with a_i^k, and for $\delta(a_i, k, t)$ we use the short hand notation $\delta(a_i^k)$ when t is clear from the context. The k^{th} occurrence of a_i processes the k^{th} packet.

We introduce the *schedule* function $\sigma : A \times \mathbb{N} \times T_{il} \rightarrow \mathbb{N}$ of the concurrent execution of components C_i, i=1..N on a processor. The schedule function returns the *start time* for each action occurrence. For $\sigma(a_i, k, t)$ the short hand notation we use is $\sigma(a_i^k)$. The *finish time* of a_i^k is $\sigma(a_i^k) + \delta(a_i^k)$.

In general, a valid schedule σ for a trace t in traceset T_{il}, satisfies the following criterion:

For each $t = s_1\, a_i^k\, b_j^l\, s_2$, $\sigma(b_j^l) \ge \sigma(a_i^k) + \delta(a_i^k)$.

In reality, when a chain executes in a cooperative environment, components execute the soonest possible given the channel and priority constraints. This leads to the schedule satisfying the following:

For each $t = s_1\, a_i^k\, b_j^l\, s_2$,

$\sigma(b_j^l) = \sigma(a_i^k) + \delta(a_i^k)$, *if* $i = j$

$\sigma(b_j^l) = \sigma(a_i^k) + \delta(a_i^k) + \Delta_{cs}(a_i^k, b_j^l, s_1)$, *otherwise.*

$\Delta_{cs}(a_i^k, b_j^l, s_1)$ is the overhead introduced by a context switch between action occurrences a_i^k and b_j^l executing after prefix s_1 of trace t. The overhead introduced by each context switch due to instruction cache misses can be calculated at design time. That is because *Theorem 6* allows us to calculate at design time the position of context switches in trace ρ. This allows the calculation at design time of the overhead introduced by a context switch due to the replacing of instructions in the instruction cache. The overhead introduced by data cache misses depends on the input data; it can be predicted using techniques as in [14].

We define the *response time* $R_{i,k}$ (Figure 3) of component C_i ($1 \le i \le N$) that processes the k^{th} packet:

$R_{i,k} = \sigma(fq_i!^k) + \delta(fq_i!^k) - \sigma(fq_{i-1}?^k)$, for all $1 \le i \le N$.

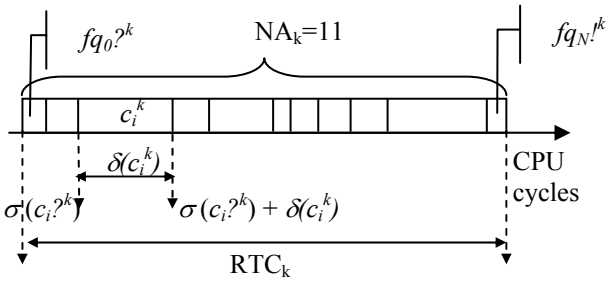

Figure 3. A schedule associated with a trace. Chain response time corresponding to NA_k

Additionally we define the *response time of the chain* (RTC) for packet k as the time counted from the moment that the packet starts being processed by the first action of C_1 until the finish time of the last action of C_N:

$RTC_k = \sigma(fq_N!^k) + \delta(fq_N!^k) - \sigma(fq_0?^k)$

We denote the number of actions occurring from the arrival of packet k (as a result of action $fq_0?^k$) until the packet leaves the chain (as a result of $fq_N!^k$) with NA_k. For instance in Figure 3, NA_k is 11.

Knowing trace ρ allows the calculation of the number of context switches (NCS (ρ)) and NA_k for any packet k in the input stream. Trace ρ, the function δ, and the cost of each context switch Δ_{cs}, allows the calculation of the
schedule function. From here, the start and finish time of each action in ρ, the response time for each component, the RTC_k for each k packet, the CPU and memory utilization can be also be calculated.

6.2 Optimizing chain response time

The response time of the chain for a packet k can be optimized when optimizing NA_k. We show in the following that NA_k is minimum when $P(C_1) = \min_{i=1..N} P(C_i)$.

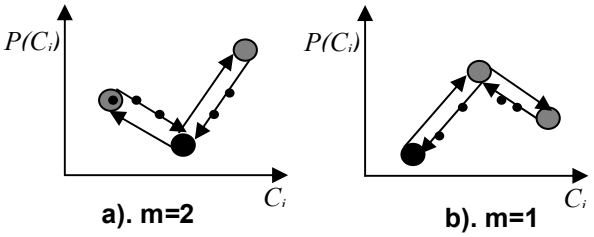

Figure 4. Priority assignment and queue occupation at the beginning of a t_{stable} iteration.

Consider the example depicted in Figure 4-a) and b). The two figures show two possible priority assignments for a chain consisting of three components. We recall that similar as before we denote with C_m the component with minimum priority in the entire chain ($P(C_m) = \min_{i=1..N} P(C_i)$). We consider $Cap(fq_i) = 2$ for all $1 \le i < N$. After calculating trace ρ for the given priority assignments, we find that in case a) (when $m>1$), for packets $k = 1..3$ that enter the chain during the initial phase: $NA_1 = 26$, $NA_2 = 36$, $NA_3 = 31$. For packets $k>3$ that enter the chain during the stable phase $NA_k = 52$.

In case b) (when $m=1$), $NA_k = 15$, for all $k \in \mathbb{N}$, $k>0$. The reason for NA_k being lower in case b) than in case a) is that in case a) packets have to "wait" for the end of the initial phase before being processed to the end of the chain, while in case b) the initial phase does not exist. In addition, in b), all packets are processed to the end of the chain immediately when they enter the chain, while in a) even during the stable phase, after entering the chain, packets must first wait for other packets preceding them in the FIFO queues in front of C_m to be processed.

We generalize these observations in the following theorem.

Theorem 13. NA_k is minimal for all $k \in \mathbb{N}$, $k>0$ when $P(C_1) = \min_{i=1..N} P(C_i)$.

Proof. We distinguish the case that $m>1$ from $m=1$.

i. $m > 1$

A. If a k^{th} packet has been inserted in the chain during the initial phase then,

$1 \le k \le \sum_{i=1}^{m-1} Cap(fq_i) + 1$ (*Corollary 8*),

and t_{init} can be written as: $t_{init} = u_1\, fq_0?^k\, u_2$. We denote $fq_0?^k\, u_2$ with w_k. In this case $NA_k = |w_k| + k * 5*N$.

Indeed, any packet k inserted in the chain during the initial phase will not be processed to the end of the chain before the initial phase is completed. Therefore, regardless at which position in the trace t_{init}, $fq_0?^k$ occurs, $|w_k|$ actions must be executed before even the first packet in the stream can be processed to the end of the chain. The length of $|w_k|$ for any packet k other than the first packet depends on the priority assignment. For $k=1$, $|w_k| = |t_{init}|$.

Once the initialisation phase is completed, packet k must "wait" for the other $k-1$ packets to be processed to the end of the chain. For each packet this happens within one iteration of the stable phase (t_{stable}), that is 5*N actions. That means that the total number of actions that occur between $fq_N!^k$ and $fq_0?^k$ is $NA_k = |w_k| + k*5*N$ (we counted the iteration for processing packet k itself too).

B. Any packet $k > \sum_{i=1}^{m-1} Cap(fq_i) + 1$ arrives during the stable phase. Each packet k arrives during the iteration of t_{stable} that processes packet $k - (\sum_{i=1}^{m-1} Cap(fq_i)) - 1$ to the end of the chain (more precisely during the trace t_L).

We express t_L as $s_1\, fq_0?^k\, s_2$, and denote $fq_0?^k\, s_2$ with v_k. The entire iteration can be written as:

$fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ \ t_L\ fq_m!\ t_R \quad =$
$fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ \ s_1\, fq_0?^k\, s_2\ \ fq_m!\ t_R =$
$fq_{m-1}?\ eq_m?\ c_m\ eq_{m-1}!\ \ s_1\ v_k\ \ fq_m!\ t_R.$

This implies that $|v_k| + |t_R| + 1$ actions are executed from the arrival of packet k (as a result of $fq_0?^k$), until this iteration is completed. After these actions, packet k must "wait" for $\sum_{i=1}^{m-1} Cap(fq_i)$ packets to be processed to the end of the chain. When counting also the iteration that processes packet k itself, we find that

$$NA_k = |v_k| + |t_R| + 1 + (\sum_{i=1}^{m-1} Cap(fq_i) + 1) * 5*N$$

$$= |v_k| + (N-m)*5*N + 1 + (\sum_{i=1}^{m-1} Cap(fq_i) + 1) * 5*N$$

ii. $m = 1$.

When $P(C_1) = \min_{i=1..N} P(C_i)$, according to *Corollary 11*, the initial phase is eliminated completely, each packet k arrives during the k^{th} iteration of t_{stable} and does not have to wait for other packets in front of it in the queues before it is processed. From here follows that $NA_k = 5*N$.

When comparing the formulas obtained for NA_k in cases *i.* and *ii.*, it follows directly that for $m=1$, NA_k is minimal. □

7. Conclusions

In this paper we have presented a model for the dynamic behavior of linear media processing chains executing in a cooperative environment. We express the behavior of the chain as a trace ρ of the actions of the components that make up the chain. We have formally proven that the trace ρ becomes repetitive (the stable phase) after a finite prefix (the initial phase) and we have shown that trace ρ can be calculated at design time. This approach allows the calculation and optimization of the capacities of the queues between components, of the initial phase, of the number of context switches, and of the response time of individual components and the entire chain.

The repetitive nature of the chain is an important property that also makes reasoning about composition of chains much easier. Designers have only to reason in terms of patterns of execution at the level of the chain instead of reasoning about the individual behavior of components within the whole system. This approach also makes systems open in the sense that the effect of inserting (or withdrawing) components from a chain can be rigorously predicted and controlled.

Future work will tackle chains including components with periodic behavior and components whose behavior is influenced by the input stream. Future contributions will deal with variable computation times for tasks, and chains with timing requirements. We will also investigate other chain topologies such as branching chains and chains composition. Finally we plan to study the applicability of this approach on streaming systems executing on multiprocessor platforms.

8. References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture,* John Wiley & Sons Ltd., 1996.

[2] C.W. Mercer, S. Savage, H. Tokuda, "Processor Capability *Reserves:* Operating System Support for Multimedia Applications", *International Conference on Multimedia Computing and Systems,* pp.90-99, 1994.

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

[4] C.C. Wust, E.F.M. Steffens, R.J. Bril, W.F.J. Verhaegh, "QoS Control Strategies for High-Quality Video Processing", *Euromicro Conference on Real Time Systems,* 2004.

[5] Zhun Zhong and Yingwei Chen, "Scaling in MPEG-2 Decoding Loop with Mixed Processing", *Digest of Technical Papers IEEE International Conference on Consumer Electronics (ICCE)*, June 2001, pp. 76-77.

[6] James Lyle Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice Hall, 1981.
[7] Angel M. Groba, Alejandro Alonso, Jose A. Rodriguez, Marisol Garcia-Valls. "Response Time of Streaming Chains: Analysis and Results". *Proceedings of the 14th Euromicro Conference on Real-Time Systems,* 2002, pp 182-192.
[8] M. González Harbour, M. Klein and J. Lehoczky. "Fixed priority scheduling of periodic tasks with varying execution priority. " *Proceedings of the IEEE Real time Systems Symposium* , *pp. 116-128*, December 1991.
[9] S.S. Bhattacharyya, P.K. Murty, and E.A. Lee, Software Synthesis from Dataflow Graphs Kluwer Academic Press, 1996.
[10] M.K. Klein, T.H. Ralya, B. Pollac, R.Obenza, M. Gonzalez-Harbour, *A practitioner's Hanbook for Real-Time Analysis: Guide to Rate monotonic Analysus for Real-Time Systems,* Kluwer Academic Publishers, 1993.
[11] S. Goddard, "Analyzing the Real-Time Properties of a Dataflow execution Paradigm using a Synthetic aperture Radar Application", *Proc. IEEE Real-Time Technology and Applications Symposium*, June 1997, pp. 60-71.
[12] R. Gerber, S. Hong, "Semantic-based compiler transformations for enhanced schedulability", *Proc. IEEE Real-Time Systems Symposium* 1993, pp. 232-242.
[13] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.
[14] S. Peng, "MPEG2 Decoding with Graceful Degradation," *Proceeding of Philips Digital Video Technologies 2000 Workshop*, 2000.