Analysis of a Time-driven Chain of Dependent Components

M.A. Weffers-Albu[1], J.J. Lukkien[1], P.D.V. van der Stok[1,2]
[1]*Technische Universiteit Eindhoven,* [2]*Philips Research Laboratories*
{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 tasks communicating via bounded buffers. The last task 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. Additionally we derive design guidelines, supporting the goal of optimizing the resources needed by the chain, achieved by minimizing context-switching overhead and buffer sizes.

1. Introduction

We consider the problem of processing a media stream by an application consisting of a chain of given off-the-shelf processing components, executed on a scarce-resource embedded platform. The essential requirement on the physical platform is cost-effectiveness, and the requirement on the application is robustness. These requirements lead to minimizing the resources made available to the application to the limit that it remains robust. In the context of our work the criteria for robustness for an application are meeting the application's real time constraints. The real-time constraints come from the fact that media processing applications must display audio/video information at a certain rate to avoid video/audio artefacts.

To ensure that the chain meets its timing constraints, it is provided with a *guaranteed resource budget*. 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 response times, to minimize buffer sizes and context switch overhead, and to reason about chain composition.

Our research aims at providing an underlying theory that helps engineers to reason rigorously about system behavior and associated resource needs. In [2] 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 last component has periodic behavior.

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 which can be calculated at design time. Given the computation times for each action in the trace, the associated schedule can be derived as shown in [2]. From here, 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, we calculate the necessary and sufficient capacities for each buffer in the chain.

This paper is organized as follows. Section 2 gives insight into the architecture style of our system. Section 3 presents a characterization of a single streaming chain execution in which the last component has periodic behavior. Related work is presented in Section 4 and Section 5 is reserved for conclusions. We rely on [2] for the more complete description of context and theory and include just enough here to make the paper self-contained.

2. 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) [4] 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 (Figure 1). 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.

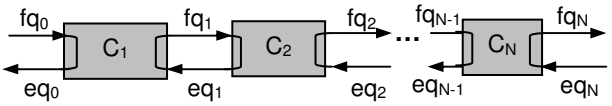

Figure 1. Chain of components.

Each component C_i $(1 \le i \le N)$ 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. We denote with $L(q)$ the number of packets in queue q. This is expressed as follows:

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

We use a simple syntax for the program text of the components using repetition ('while'), sequential composition (';') and basic statements (*actions*). The set of basic statements of a component C_i is called its alphabet $A(C_i)$. 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. We denote the i^{th} occurrence of an action a in a trace with a^i.

The behavior of C_i, $1 \le i < N$, 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.

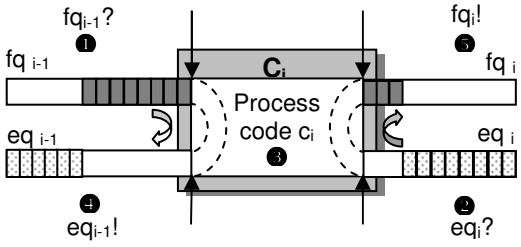

Figure 2. A basic streaming component C_i.

The syntax for the execution of a component C_i $(1 \le i < N)$ and the trace set are:

C_i: *while (true) do*
{ *receive(fq_{i-1}, p); receive(eq_i, q);*
process_func_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.

The last component in the chain (C_N) has similar behavior as the rest with respect to the operations on the queues. However this component has periodic behavior meaning that any two consecutive iterations of its loop are executed at one period T_N distance in time. To specify this behavior we introduce an additional statement called *delay($i*T_N$)* where $i*T_N$ refers to an absolute time. The informal interpretation of *delay($i*T_N$)* is that it delays the next iteration of component C_N until time is at least $i*T_N$. The computation time of the delay action is 0. The syntax for the execution of component C_N and the traceset are:

C_N: *i:=0;*
while (true) do
{ *i:=i+1;*
receive(fq_{N-1}, p); receive(eq_N, q);
process_func_c_N(p, q);
send(eq_{N-1}, p); send(fq_N, q);
delay($i*T_N$); },

with traceset $Tr(C_N)$:

$\{i = 0 \ (inc(i) \ fq_{N-1}? \ eq_N? \ c_N \ eq_{N-1}! \ fq_N! \ d(i*T_N))^{\omega}\}$.

We call this component *time-driven* component. Notice that we denote a *delay* action in the trace by *d*.

All traces presented above are infinitely repetitive and all actions that compose the traces are atomic. In general, the set of states during an execution is characterized by the prefixes of its traceset, the *prefix closure*. For a traceset T we denote this set of states by $St(T)$.

Finally, we define the projection of a trace t to 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. Characterization of chain execution

Our system (chain) consists of N communicating components executing in parallel on the physical platform. As we have shown in [2], 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_N, and the priority assignment to components. As in [2], 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. The schedule function is defined as $\sigma: St(T_{il}) \rightarrow \mathbb{N}$ where for any state s from $St(T_{il})$, $\sigma(s)$ returns the finishing time of state s in a trace t of T_{il}.

As a first step, from all traces in T_{il} we restrict ourselves again to the *channel consistent traces* that satisfy the channel invariant *Scc*. The set obtained by

imposing *Scc* on all traces from T_{il} is called T_{cc}:

$Scc(t)$: $(\forall s \in Pref(t), fq_i, eq_i (1 \le i < N)$:

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

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

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 $\mathcal{A}(C_i)$ 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}.

All schedules associated with each trace in T_{cc} satisfy the soundness criterion ($\forall sa \in St(T_{cc})$, $\sigma(sa) \ge \sigma(s) + \delta(a)$, where $\delta(a)$ is computation time of action a). The periodic behavior of C_N (implemented with the aid of the delay action), induces a property on the schedules associated with all traces in T_{cc} of this chain:

$$\sigma(s\, d(i * T_N)^i) \ge i * T_N, \quad \forall \sigma \wedge s \in St(T_{cc}) \qquad (1)$$

The meaning of (1) is that when a delay action ($d(i * T_N)$) follows a state s ($s \in St(T_{cc})$) at a time earlier than $i*T_N$, the delay action advances the time until $i*T_N$. When the delay action follows s at a time equal or later than $i*T_N$, the *delay* action has no effect. We take into account this restriction in the process of deciding the next action when constructing a trace by imposing the following constraint:

$S\sigma c(u)$:$(\forall s \in Pref(u), d(i*T_N) \in A(C_N), v \in A^{\omega}$:

$$u = s\, d(i*T_N)^i\, v: \sigma(s) \ge i * T_N \vee Act(s) = \varnothing),$$

where $Act(s)$ is the set of actions (other than *delay*) that are *ready to run* in state s. Predicate $S\sigma c$ imposes that a delay action $d(i*T_N)^i$ can follow a state s in a trace u, only if $\sigma(s)$ is later in time than time $i*T_N$ specified in $d(i*T_N)^i$, or there are no actions (other than *delay* actions) that are ready to run. $S\sigma c$ implies that a delay action can only execute at an earlier time than $i*T_N$ when there are no other actions (other than delay actions) that are ready to run. From here follows that in this case the time interval $i*T_N - \sigma(s)$ 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 Sσc:

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

Predicate $S\sigma c$ implies that if an occurence $i+1$ of an action from $A(C_N)$ follows a state $v \in St(T_{\sigma c})$ in a trace $u \in T_{\sigma c}$ then $\sigma(v) > i*T_N$. This is expressed as follows:

Property 1. $(\forall v \in St(T_{\sigma c}), a \in A(C_N), va^{i+1} \in St(T_{\sigma c}) : \sigma(v) > i*T_N)$.

We are ready now to define the notion of *ready to run* and *blocked* from *time perspective* for all actions of $A(C_N)$. Any action a in $A(C_N)$ such that $sa^i \uparrow A(C_N)$ is a state of $Tr(C_N)$ and sa^i is a state of $T_{\sigma c}$ (which means that $\sigma(s) > (i-1)*T_N$) is called a *ready* action of C_N in state s. If sa^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_N is *blocked from time perspective* at a^i in state s of $T_{\sigma c}$, denoted as 'C_N **bt** a^i [in s of $T_{\sigma c}$]'. We also say that component C_N is *idle* in state s.

The eagerness criterion that imposes each action to be scheduled as soon possible, reduces the set of possible schedules associated with any trace in $T_{\sigma c}$, to just one.

Next we introduce an additional restriction on the traces of $T_{\sigma c}$ in the form of a priority assignment as shown in [2]. This translates into the invariant *Scp* that in each state of $T_{\sigma c}$ where there are multiple components *ready*, the *ready* action of the component with the highest priority is selected. Limiting $T_{\sigma c}$ according to *Scp* gives T_{pc}, the *priority consistent* traces: $T_{pc} = \{ t \in T_{\sigma c} | Scp(t)\}$. As in [2] we have:

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

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

In order to guarantee that component C_N has a periodic behavior it is necessary that C_N becomes *ready to run* every period T_N both from *time* and *channel perspective*. In the media streaming domain this is particularly important because the last component in the chain acts as an audio or video renderer and must display the video information on the TV screen at a rate adjusted to the human ear/eye. Any delay in displaying the information causes the perceived quality of service to diminish. Given the restrictions imposed by $S\sigma c$, C_N becomes *ready to run* from *time perspective* every new period. However, in order to ensure that C_N becomes ready from channel perspective every T_N, we must ensure that C_N receives every new period a new packet to process. We denote the time between the production of any 2 consecutive packets k and $k+1$ in fq_i with PR_k^i. To make sure that every new period at least 1 new full packet is produced in fq_{N-1} we impose in the rest of our study the following restriction on the full packets production rate:

$$PR_k^{N-1} \le T_N, \ \forall k \in \mathbb{N}. \qquad (2)$$

In the case of trace ρ, it is trivial to prove that PR_k^{N-1} is the sum of the computation times of one iteration of each C_i ($1 \le i < N$). The restriction expressed in (2) implies that every T_N there will be more packets produced in fq_{N-1} than there will be consumed. From here follows that after a finite number of periods T_N fq_{N-1} is filled to its capacity. When, this happens, eq_{N-1} is drained, which implies that C_{N-1} becomes blocked at action $eq_{N-1}?$. From here on, C_{N-1} is de-blocked (and therefore executes) only when C_N produces an empty packet in eq_{N-1}, which happens only once every T_N. This situation makes it that from here C_{N-1} will have an induced periodical execution (with period T_N) due to its dependency on C_N to release an empty packet in eq_{N-1}. By repeating the reasoning above for the sub-chain composed of components C_1, …, C_{N-2} we

find again that within a finite number of actions fq_{N-2} will be filled to its capacity (PR_k^{N-2} is the sum of the computation times of one iteration of C_i $1 \leq i < N-1$, hence $PR_k^{N-2} \leq T_N$). The reasoning continues similarly until all forward queues are filled (backward queues are drained) and all components are dependent on C_N to produce one empty packet, which causes a de-blocking in cascade of components $C_1,\ldots,C_{N-1}$ (exact order of execution determined by priority assignment). All components will execute one iteration of their individual traces after which they become again blocked on their input backward queue and must wait again for C_N to produce 1 empty packet in eq_{N-1}. The fixed priority assignment to components and the fact that (2) is satisfied during each period T_N determines a repetitive execution of the system. The above reasoning leads us to the following theorem:

Theorem 3. Under the assumption that $PR_k^{N-1} \leq T_N$, $k \in \mathbb{N}$, the pipeline system in which the last component executes periodically assumes a repetitive behavior (called stable phase denoted by trace t_{stable}) after a finite initial phase (t_{init}). The complete behavior is characterized by the unique trace ρ:

$\rho = t_{init}\ (inc(i)\ fq_{N-1}?\ eq_N?\ c_N\ eq_{N-1}!\ t_L\ fq_N!\ d(i * T_N))^{\omega}$

where t_L is the subtrace recording the parallel execution of components $C_1,\ldots, C_{N-1}$.

As in [2], trace ρ can be calculated by choosing in each state the *ready* action of the component with the highest priority. Knowing the trace allows to calculate the number of context switches. The eager schedule of the unique trace can also be calculated provided that the computation times of each action processing an input stream are known. This renders the start and response times for individual tasks and the response time of the chain.

Important to note is that because of (2), regardless of its priority, component C_N has the same effect on the stable phase trace of this chain, as a data-driven component with minimum priority has in the case of a chain composed of only data-driven components (described in [2]). Owing to this fact, we find that corollaries addressing the stable phase of a chain composed of only data-driven components (with C_N having the lowest priority) hold in this case as well. From here we deduce that the number of context switches during the stable phase can be reduced by assigning priorities as $P(C_1) < P(C_2) < \ldots < P(C_{N-1})$ and $Cap(fq_i) = 2$ *for all i,* $1 \leq i < N-1$ (Theorem 12, [2]). The minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 9, [2]). Also, we find that the response time of the chain cannot be improved because assigning the minimum priority to C_1 as suggested in [2] does not change the influence of the time-driven component C_N.

4. Related work

Several attempts have been made to analyze message passing, streaming systems. Closely related work in [3] presents an analysis method allowing the calculation of the worst–case response time of multiple TSSA video streaming chains based on the canonical form of the chains.

Goddard [5] studies the real-time properties of PGM dataflow graphs, closely resembling our media processing graphs. Given a periodic input and the dataflow attributes of the graph, exact node execution rates are determined for all nodes. 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. Finally, in [6] the authors focus on the fixed priority scheduling of periodic tasks decomposed into serially executed sub-tasks but no intermediate buffers are considered.

5. Conclusions

We have presented a characterization of the dynamic behavior of a media streaming chain in which the last component is time-driven and the rest of components are data-driven. We have shown that after a finite prefix (initial phase) the trace recording the execution of the chain becomes repetitive (stable phase). Additionally we have shown that the time-driven component has the same influence on the overall execution of this chain as a data-driven component with minimum priority has on a chain of only data-driven components. This reduces the analysis of this time-driven system to be identical to that of the data-driven system in [2].

References

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

[2] M.A. Weffers-Albu, J.J. Lukkien, E.F.M. Steffens, P.D.V. v.d. Stok. On a Theory of Media Processing Systems Behavior, with applications, *ECRTS 2006,* in press.

[3] Angel M. Groba, Alejandro Alonso, Jose A. Rodriguez, Marisol Garcia-Valls. Response Time of Streaming Chains: Analysis and Results. *ECRTS 2002, pp 182-192.*

[4] G. v. Doren, B. Engel. Streaming in Consumer electronic Products Beyond Processing Data. In Dynamic and Robust Streaming in and betweenConnected Consumer Electronic Devices, Philips Research Series, Springer, 2005.

[5] S. Goddard, "Analyzing the Real-Time Properties of a Dataflow execution Paradigm using a Synthetic aperture Radar Application", *RTAS 1997*, pp. 60-71.

[6] Michael Gonzales Harbour, Mark H. Klein, John P. Lehoczy. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. *IEEE Transactions on Software Engineering,1994, Vol. 20, pp. 13-28.*