A Characterization of Streaming Applications Execution

M.A. Weffers-Albu[1], J.J. Lukkien[1], P.D.V. v.d. Stok[1,2]

[1]Technische Universiteit Eindhoven, [2]Philips Research Laboratories

Abstract

In this article we provide a model for the dynamic behavior of a single video streaming chain, by formulating a theorem describing the stable behavior. This stable behavior is characterized in terms of the elementary actions of the components in the chain, from which standard performance measures (such as activation time, response time and resource utilization) follow. From this we derive corollaries, which give guidelines for the design of the chain, while targeting optimization of resource utilization and minimizing context-switching overhead.

1. Introduction

We consider the problem of processing a video stream by an application consisting of a chain of given processing components, on a scarce-resource embedded platform. The essential requirement on the platform is cost-effectiveness, leading to minimizing the resources made available to this application. The requirement on the application is robustness. In terms of the real-time tasks that compose the application, this leads to the requirement of predictability of timing behavior and (shared) resource use. In order to support the design of the application (mainly the mapping of such an application onto an execution platform) a good model of its run-time behavior is needed such that timing and resource utilization can be accurately predicted. In addition, the model can be used to control that behavior.

The parameters we want to predict and control are attributes such as activation time (AT) and response time (RT) of tasks and of the chain as a whole, as well as resource utilization (RU) for CPU, memory and bus. Predicting and controlling the RT of each task in the chain is important for the prediction and control of the response time of the entire chain. This article presents a first step in this work by characterizing the execution of a single video streaming chain by formulating a stable-phase theorem. Subsequent corollaries provide guidelines for design aiming at improving the resource utilization and minimizing the context switching overhead. We adopt as an important measure for the overhead the number of context switches (NCS).

The article presents our execution model in section 2, and a characterization of a single streaming chain execution in section 3. Related work is presented in section 4 and section 5 is reserved for conclusions.

2. TriMedia Streaming Software Architecture

This work has been done in collaboration with the *Multi-Resources Management* project at Philips Research Laboratories Eindhoven where we study the TriMedia Streaming Software Architecture (TSSA). TSSA is an instantiation of the *pipes and filters architecture style* [6] and it provides a framework for the development of real time audio-video streaming applications executing on a single TriMedia processor [1]. A media processing application is described as a graph in which the nodes are software components that process data, and the edges are finite queues that transport the data stream in packets from one component to the next (Figure 1).

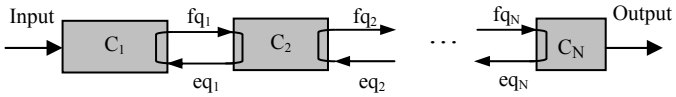

Figure 1 - Chain of components.

Each component C_i ($1 \leq i \leq N$) has an associated task to which a unique fixed priority is assigned (fixed priority scheduling) and the tasks execute as long as input is provided. Every connection between two components is implemented by two queues. One queue (called *full queue*) carries *full* packets containing the data to be sent from one component to the next, while the second queue (the *empty queue*) returns *empty* packets to the sender component to recycle packet memory. The empty packets are returned in order to signal that the data has been received properly and that the memory associated with the data packet may be reused. Each component C_i ($1< i <N$) has two input queues (a full queue fq_{i-1} and an empty queue eq_i) and two output queues (a full fq_i and an empty queue eq_{i-1}) (Figure 1). C_1 and C_N are connected to their neighbors by only two queues. C_1 has one input empty

queue eq_1 and one output queue fq_1. C_N has one input full queue fq_{N-1} and one output queue eq_{N-1}.

A typical *execution scenario* of C_i (denoted by $E(C_i)$) (Figure 2) is the following: the component gets 1 full packet from the input full queue ($fq_{i-1}?$) ❶, then gets 1 empty packet from the input empty queue ($eq_i?$) ❷, performs the processing (α) ❸, recycles the input packet by putting it in the output empty queue ($eq_{i-1}!$)❹ and, finally, the result of processing is put in the output full queue ($fq_i!$) ❺.

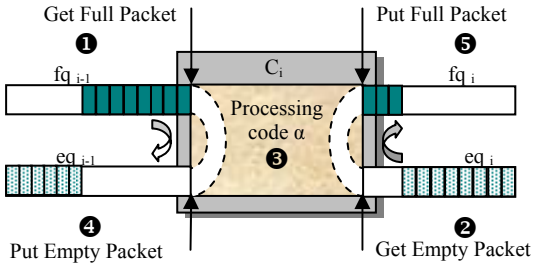

Figure 2. A basic streaming component.

Note that the semantics of the $fq_{i-1}?$ and $eq_i?$ operations is that the number of packets in the fq_{i-1} and respectively eq_i queue is decreased by one. Also, the semantics of the $eq_{i-1}!$ and $fq_i!$ operations is that the number of packets in the $eq_{i-1}!$ and respectively $fq_i!$ queue is increased by one.

3. A characterization of chain execution

In the current article we will focus on the case of a single linear streaming chain (Figure 1) consisting of event-driven components with fixed worst-case execution times and executing in a cooperative environment. The latter means that the environment will always provide input and always accept output. In general, a chain will not exist in isolation, but composed with other chains, the components can have different execution scenarios, and the environment is not always cooperative. As we mentioned in the previous section, the analysis we present in this paper is a first step in analyzing this type of streaming systems, step we consider necessary before tackling more complex cases. We also observe that the (environment) input and output of the chain are different than the connections (via buffers) between components because while C_1 and C_N will never block on the environment input and respectively output, they can block on their input empty and respectively full queue.

The initial situation of the chain is that all full queues are drained (which implies that all empty queues are filled to their full capacity). At any time the task with the highest priority that has enough input to run will execute. We will use the following notations: P_i refers to the priority of component C_i, L(Q) denotes the number of packets in queue *Q*, |Q| denotes the capacity of queue *Q*, C_i **b** Q means that component C_i is blocked on queue *Q*. C_M denotes the component with minimum priority in the chain.

In order to characterize the system behavior we list invariant properties that we derive from the components behavior and from the priority assignment. Assuming the capacities of empty queues are identical to the capacities of the corresponding full queues we derive from the components behavior:

Property 1 – $\forall\ i,\ 1< i \leq N,\ C_i$ **b** eq_{i-1} is not possible.

Property 2 – $\forall\ i,\ 1\leq i < N,\ C_i$ **b** fq_i is not possible.

Property 1 and 2 state that blocking at the output of a component is not possible.

Whenever a component executes, it can do so only if higher priority components are blocked. Consider a descending chain of priorities starting from the input side of the chain. Whenever the last component in this chain executes, the other ones are blocked. Because of the cooperative environment all components (except the last one) are blocked on reading from the empty queue. This situation can be generalized to a minimal priority component.

Lemma 3 - When C_i is such that $\forall$ j, $j<i$, $P_j > P_i$ and C_i is executing in α, then C_j **b** eq_j.

A similar result holds for the end of the chain.

Lemma 4 - When C_i is such that $\forall$ j, $j > i$, $P_j > P_i$, and C_i is executing in α, then C_j **b** fq_{j-1}.

For component C_M the situation is special in that both lemmas 3 and 4 apply. As a result, whenever C_M executes the remainder of the chain is blocked. In addition, whenever C_M de-blocks one of its neighboring components, the components in the corresponding half-chain will take over, execute one time their execution scenario after which they will return to their blocked state again. The sequence of actions in doing this is completely determined by the priority assignment. Therefore, the behavior of the system can be described as the interleaving of the behavior of C_M with these left and right half-chain behaviors.

Corollary 5 - When C_M is in α, $\forall$ i: $1 \leq i < M$, $L(fq_i) = |fq_i|-1 \wedge L(eq_i)=0 \wedge \forall$ i: $M \leq i < N$, $L(fq_i) = 0 \wedge L(eq_i)=|eq_i|$.

Stable Phase Theorem - Let $C_1, C_2, C_3, \ldots, C_N$ be a chain of event-driven components communicating through a set of queues as in Figure 1. Provided that the input is sufficiently long, the execution of the components in the chain will adopt a repetitive pattern (during which the chain is in a stable phase) in a finite number of steps (initialization phase). The repetitive pattern of execution is:

$fq_{M-1}?$; $eq_M?$; α; $eq_{M-1}!$; $E(S_L)$; $fq_M!$; $E(S_R)$,

where $S_L=\{C_1, \ldots, C_{M-1}\}$, $S_R=\{C_{M+1}, \ldots, C_N\}$ and $E(S_L)$ and $E(S_R)$ are the mentioned combined executions of components in sub-chain S_L and sub-chain S_R respectively. Note that the repetitive pattern of execution depends on the execution scenario of the components. In

future work we will consider chains composed of components with different execution scenarios and we will show again how the different execution scenarios influence the pattern of execution.

Property 6 - The length of the initialization phase in number of execution steps is $\sum_{j=1}^{M-1} \sum_{i=j}^{M-1} | eq_i |$.

Corollary 7 - The length of the initialization phase can be reduced by reducing the length of all queues connecting the components preceding C_M in the chain to the limit necessary to prevent deadlock.

Corollary 8 - The minimum queue length sufficient for each of the queues in the chain is 1.

Corollary 9 - The initialization phase can be eliminated completely by assigning to C_1 the minimum priority.

Corollary 10 – In the stable phase the execution of all components is driven by the execution of C_M.

Corollary 11 - At the beginning of the stable phase $\forall$ i $\neq$ M, C_i is blocked while C_M is ready-to-run.

Corollary 12 – NCS, AT, RT of all tasks involved, RT for the entire chain, and CPU utilization during one execution of the pattern can be calculated by reconstructing the first execution of the pattern considering that in the beginning of the stable phase the states of all component tasks are as described in Corollary 11.

The algorithm for reconstructing the execution of the pattern is described in [2].

The following corollary provides guidelines on how to assign priorities to tasks in the chain such that the NCS is minimized. We are interested in minimizing the NCS because we want to minimize the overhead.

Corollary 13

1. The minimum NCS during initialization phase can be achieved when $C_M = C_1$.
2. The minimum NCS during one execution of the pattern can be achieved either when:
   a. $P_1<P_N<P_{N-1}<\ldots<P_2$,
   b. or when $P_N<P_1<P_2<\ldots<P_{N-1}$.

4. Related work

Closely related work in [3] and [4] also considers an execution model for video streaming chains inspired by TSSA. Both articles present an analysis method allowing the calculation of the worst–case RT of multiple video streaming chains based on the canonical form[1] 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. In contrast, in our analysis we can deal with variable execution times and take the overhead of context switching into account. In addition, we provide guidelines for the optimization of RU.

The execution model used in [3] and [4] presents only one buffer linking two consecutive components in a chain. Although we expect that a similar stable phase theorem could be stated also for this execution model, the corollaries derived would differ because of the different execution model used.

Finally, in [5] the authors focus on the fixed priority scheduling of periodic tasks decomposed into serially executed sub-tasks but no intermediate buffers are considered.

[1] The canonical form of a chain CN composed of N tasks, can be seen as a new chain CN' composed of N' tasks (N' $\leq$ N) where the priorities of the tasks do not decrease from input to output.

5. Conclusions

We have presented a characterization of the dynamic behavior of a video streaming chain composed of event-driven components. The presented theorem and lemmas show that after a finite initialization phase streaming chains reach a repetitive pattern of execution. This repetitive pattern is exploited further in predicting attributes such as activation time, response time of individual tasks, response time of the entire chain, the number of context switches and resource utilization. Additional corollaries provide guidelines for the design, while targeting optimization of resources and minimizing context-switching overhead.

Acknowledgements

We are thankful to Reinder Bril, Liesbeth Steffens, Clara Otero-Perez, Laurentiu Papalau, Giel van Doren and Dietwig Lowet for their helpful comments and suggestions during the process of reaching the present results.

6. References

[1] P.N. Glaskowski. Philips advances TriMedia architecture – New CPU64 core aimed at digital video market. *Microdesign Resources, 1998, vol.12, no 14, pp. 33-35.*

[2] M.A. Weffers-Albu, P.D.V. v.d. Stok, J.J. Lukkien. NCS Calculation Method for Streaming Applications. *Proceedings of the 5th PROGRESS workshop on embedded systems,2004,pp.3-9.*

[3] 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.*

[4] Angel M. Groba, Alejandro Alonso. Response Time Analysis of Periodic Chains. *The 22nd IEEE Real-Time Systems Symposium, 2001,Work in Progress, pp. 29-32.*

[5] Michael Gonzalez Harbour, Mark H. Klein, John P. Lehoczy. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. *Proceedings of the IEEE Real-Time Systems Symposium, 1991, pp.116-128.*

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