Towards a Characterization of Real Time Streaming Systems

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, number of context switches and resource utilization) follow. From this we derive corollaries that give guidelines for the design of the chain, which target improving the end-to-end response time of the chain and the optimization of resources by minimizing context-switching overhead and buffer sizes.

1. Introduction

We consider the problem of processing a video 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 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, and the absence of deadlock during its execution.

The system resources we aim to minimize are the amount of processor cycles (λ), number of memory bytes (μ) and bus bandwidth (φ). In order to evaluate the overall cost on the system when minimizing one of the resources, we consider a cost function having as components the three resources above to which weights are attached in order to differentiate their importance for the designer:

$$F(\lambda, \mu, \varphi) = w_1*\lambda + w_2*\mu + w_3*\varphi.$$

Our approach is to optimize the cost-effectiveness of the application on the platform by minimizing the cost function (while meeting robustness criteria) during the design phase of the chain. In terms of the real-time tasks that compose the chain, this leads to the requirement of predictability of timing behavior and (shared) resource use. In order to support the design of the chain a good model of its run-time behavior is needed such that timing and resource utilization can be accurately predicted. In this sense we formulated a stable-phase theorem, properties and corollaries that characterize the execution of this type of streaming chains. Moreover, some corollaries provide guidelines for the design of the chain which can be used to control its behavior.

Concerning the optimization of λ and φ used by the application, we studied the optimization of the overhead of processor cycles and the bus load due to context switches which can be influenced at chain design level. A context switch happens when an executing task loses the CPU resource due to blocking, preemption or task completion and the processor is allocated to another task. The TriMedia chip [1] on which we perform our experiments, has two small cache memories (instruction cache and data cache) which means that in the process of CPU reallocation, at least part if not all of both the instruction and data lines of the first task are replaced by the instructions and data of the second task. This process implies accessing the caches and the main memory via the interconnecting bus thus involving extra of processor cycles (overhead) and bus load for each context switch. For these reasons we adopted as a measure for the overhead the number of context switches (NCS) and therefore by minimizing NCS we reduce λ and φ. Another relevant aspect is the cost associated with each context switch which is dependent on the cause that lead to it (blocking, preemption, tasks completion). For this reason it is important to predict the activation time (AT) of each of the tasks involved because AT indicates the occurrence of a context switch and the cause of it. We provide guidelines in the form of corollaries showing how the NCS and AT can be predicted for the execution of an application and what can be done to minimize NCS.

The μ used by the application consists of the sum of the sizes of queues that connect the components. For the optimization of μ we show what is the minimum size of queues sufficient for the application in order to avoid deadlocks and meet timing constraints.

With respect to meeting robustness criteria, the latency (end-to-end response time) of the chain is important to predict and control. The response time (RT) of each task

in the chain is important for the prediction and control of the end-to-end RT of the entire chain. We impose timing constraints for the chain although in this article we present the case where the streaming chain does not have periodic triggers at input or output because in further work we will consider this situation and then the results concerning the latency of this chain will be used. In this sense, the work presented here is only a first step which although more theoretic is very relevant for the future analysis on more complex, realistic situations.

Finally, in order to assist in the decision regarding which of the system resources is more important to optimize (deciding the weights in the cost function) we provide means to predict the resource utilization (RU) for CPU, memory and bus.

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. Execution Model

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 [7] and it provides a framework for the development of real time audio-video streaming applications executing on a single TriMedia processor. 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).

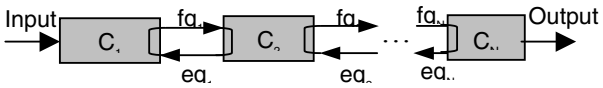

Figure 1 - Chain of components.

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. The capacity of the full queue is equal to the capacity of the empty queue.

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 (FP) from the input full queue (fq_{i-1}?) ❶, then gets 1 empty packet (EP) from the input empty queue (eq_i?) ❷, performs the processing (α) ❸, recycles the input packet from fq_{i-1} 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.

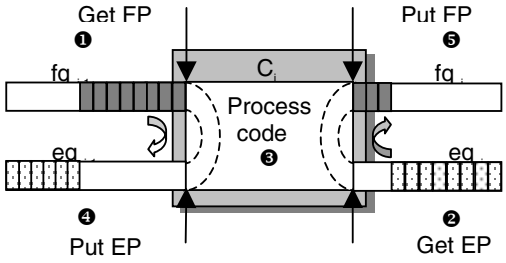

Figure 2. A basic streaming component.

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 executing in a cooperative environment. The latter means that the environment will always provide input and always accept output. 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.

Property 1 – $\forall$ i, $1 < i \leq N$, $\neg C_i$ **b** eq_{i-1}.

Property 2 –$\forall$ i, $1 \leq i < N$, $\neg C_i$ **b** fq_i.

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. The situation presented above is generalized for any priority assignment in Lemma 3 and Lemma 4 below. The two lemmas state that when a component with local minimum priority executes in α, all components preceding it in the chain, are blocked on their input empty queues (Lemma 3) and respectively all components succeeding it in the chain are blocked on their input full queues (Lemma 4).

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.

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}.

This situation can be generalized to a minimal priority component C_M. 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 system 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. These considerations have lead to the Stable Phase Theorem formulated below:

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:

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

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.

Another result (Corollary 5) derived from Lemma 3 and 4 shows the state of all queues in the system when C_M executes in α. The reason for $L(fq_i) = |fq_i|-1$ for each full queue preceding C_M, is that when C_M is in α, one packet is already inside of the component that takes input from the full queue. This situation is induced by the order of elementary actions in the execution scenario of each component (section 2).

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: $M \leq i < N$, $L(fq_i) = 0 \wedge L(eq_i)=|eq_i|$.

Corollary 6 and Corollary 7 are directly deduced from Lemma 3 and 4 above when applied for C_M and they are key in calculating the pattern of execution for the components in the chain regardless of what the priority assignment to the components is. By knowing what the repetitive pattern of execution is, performance attributes such as NCS, AT, task and chain RT and RU can be also calculated as well as shown in Corollary 8. Note that if the components have fixed execution times task and chain RT, as well as RU can be calculated precisely. If components have variable execution times, then worst-case values for the task and chain RT and for RU can be provided.

Corollary 6 – In the stable phase the execution of all components is driven by the execution of C_M.

Corollary 7 - At the beginning of the stable phase $\forall$ i $\neq$ M, C_i is blocked while C_M is ready-to-run.

Corollary 8 – NCS, AT, RT for each task, chain RT (latency), and RU for CPU and bus 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 7. The algorithm for reconstructing the execution of the pattern is described in [2].

Property 9 below shows the length of initialization counted in number of execution steps. Immediate consequences of this property are used to provide guidelines on how to reduce the initialization phase in Corollary 10 and 11. Corollary 7 shows that the initialization phase ends when C_M executes for the first time. Also the execution pattern in the Stable Phase Theorem shows that the last component in the chain produces its output after C_M executes. Therefore reducing the initialization phase leads to reducing the chain latency for the first packet in the stream.

Property 9 - 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 10 - 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 11 - The initialization phase can be reduced by choosing a smaller M. For M=1 (C_1 has minimum priority) the initialization phase is eliminated completely.

Corollary 12 shows the minimum sufficient queue capacity (thus memory) for each of the queues interconnecting event-driven components as described in section 2.

Corollary 12 - The minimum queue capacity sufficient for each of the queues in the chain is 1.

Corollary 13 provides guidelines on how to assign priorities to tasks in the chain such that the NCS is minimized. Point 1 is a direct consequence of Corollary 11, showing how can the initialization phase be reduced completely. Moreover, point 2.a presents a priority assignment that eliminates all context switches caused by preemptions during the stable phase. The initialization phase is eliminated as well.

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 $P_N<P_1<P_2<\ldots<P_{N-1}$ with $\forall$ i: $1 \leq i < N-1$, $L(fq_i)=2$.

The priority assignment suggested at point 2.b implies that all context switches caused by preemptions were eliminated, but there will exist an initialization phase. Moreover, in this setting, had the lengths of the queues been 1 (Corollary 12), additional context switches would have occurred due to the blocking on the input full queues of components $C_2,\ldots, C_{N-1}$. We notice in this case that regarding the cost function introduced at the beginning of the article, λ and φ are reduced at the cost of μ.

A further conjecture that we can formulate without proof for now is that the chain latency for any packet in the input stream can be improved when $P_1<\ldots<P_N$. Our reasoning is that due to this priority assignment every packet is passed on to the next component as soon as it has been produced without waiting until the sender component becomes blocked before it is taken by the receiver component. We note that in this case the NCS is not minimized because of the preemptions that occur due to this priority assignment. Therefore in this case we optimize the chain latency at the expense of more context switches.

4. Related work

Closely related work in [3] also considers an execution model for video streaming chains inspired by TSSA. The article presents an analysis method allowing the calculation of the worst–case RT 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. In contrast, in our analysis we can deal with variable execution times and take the overhead of context switching into account. Also the execution model used in [3] 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.

Another study regarding performance composition analysis on TSSA streaming systems is presented in [4]. This study gave estimated bounds on the NCS in terms of minimum and maximum values. Our analysis allows the calculation of NCS occurring during one execution of the repetitive pattern. In [5] the author analyses a single TSSA video streaming chain for which the tasks are assigned priorities in a V-shape model. The tasks at the extremities of the chain are time-driven and the intermediate buffers are not allowed to overflow. In contrast, our analysis does not assume a particular priority assignment to the tasks in the chain and all components are event-driven.

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 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, which target improving the end-to-end response time of the chain and the optimization of resources by minimizing context-switching overhead and queues sizes.

Future work will tackle the influence brought in the execution of a chain by tasks with execution deferral, periodic tasks, and tasks with different execution scenarios, executing in non-cooperative environments. Additionally we will study composition of chains where their execution is independent or dependent of each other.

6. References

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

[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.*

[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] E.G.P. van Doren, private communication, Philips Research.

[5] D.J.C. Lowet, private communication, Philips Research.

[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.*

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