NCS Calculation Method for Streaming Applications

M.A. Albu, P. v. d. Stok, J.J. Lukkien
Technische Universiteit Eindhoven, Philips Research Laboratories
E-mail:m.a.albu@tue.nl, peter.van.der.stok@philips.com, j.j.lukkien@tue.nl

Abstract – **Our contribution in the context of the "Quality of Service (QoS) in IN-home digital networks" project was focused on the QoS provided by consumer electronics terminals. In order to achieve a particular level of QoS provided by a terminal, an important issue is resource management supported by performance analysis. The work we present in this article is highlighting ways of predicting the necessary resources (e.g. CPU, memory, bus) needed by a video streaming application to provide a given level of QoS. We introduce a calculation method that involves measuring in isolation the resource needs of each of the individual streaming components, and also a performance composition analysis, which takes into account the Number of Context Switches (NCS) occurring during the execution of the application. We based our calculation for the NCS on the observation that running streaming applications, eventually adopt a pattern of execution that repeats after a specific interval of time (hyperperiod). By finding the NCS induced during a hyperperiod, we deduce the total NCS occurring during the execution of the application. The article gives a characterization of the streaming applications execution and of the component model that lie at the basis of our calculation.**

Keywords – **QoS; real-time embedded systems; streaming applications; context switches**

I. INTRODUCTION

The *QoS in IN-home digital networks* project aims at providing an integrated approach for achieving levels of Quality of Service (QoS) for systems consisting of a number of consumer electronics devices (called terminals) and a network that interconnects them. QoS, according to a recommendation provided by the ITU-T forum in Geneva 1994, is *the collective effect of service performances that determine the degree of satisfaction of the user of that service.* Our alternative definition of the term indicates that QoS is *a collection of (QoS) parameters values related to functional and non-functional characteristics of the service in question, and an assessment with respect to the degree of quality (unsatisfactory, good, excellent) derived from applying assessment rules on the values of these (QoS) parameters.* Examples of QoS parameters in the context of networks can be derived from the characteristics of the network transmission: *reliability*, *delay*, *jitter, or bandwidth*. In the context of the terminals, *reliability* and *performance* are two fundamentally relevant parameters.

Our contribution to the *QoS in IN-home digital networks* is focused on providing ways for enhancing the QoS of consumer electronics terminals, which can be categorized as real-time embedded systems. The reliability and performance of such systems are strongly related to their predictability, and as such, one way of improving the afore-mentioned QoS parameters is to improve the predictability of the systems in discussion.

Nevertheless, real-time embedded systems are notorious for their challenge in achieving a predictable execution. In our case the challenge is induced by the scarcity of resources provided by the platforms on which the real-time systems are implemented. The afore-mentioned resource limitation leads to resource sharing between the building *components* of the systems under discussion. That, combined with requirements of high level resource utilization raises difficult questions in terms of knowing which component will hold the system resources and until when, and whether all components will finish their tasks before their deadlines. To make

matters worse, the complexity of the analysis grows with the ever-increasing complexity of multi resource, multiprocessing real-time systems [1]. If the challenges above are not properly answered, the performance of the entire system as well as its reliability may suffer to a great extent.

A first answer to the above challenges is to incorporate performance prediction and analysis in the early stages of system architecture and design. That insures that the design of the system is aware of the level of performance to be expected on a particular platform for each of the individual components as well as for the system as a whole. Another advantage of using performance analysis earlier on is that the design will be based on strategies known to maximize performance that implies a consolidated understanding of how the system as a whole exploits the resources of the platform.

Research activities focusing on early performance prediction of software architectures were conducted in the context of the AIMES project [2] of Eindhoven University of Technology. The proposed method employs both structural and stochastic modeling techniques to those parts of the system that remain unchanged for a long time with a statistical approach and those that evolve rapidly with an analytical approach.

Other approaches to performance prediction [3], [4] use queuing network models derived from the structural description of the architecture.

Nevertheless, using performance analysis in the early phases of design is more difficult if a system is built of independent components provided by other parties, as it is progressively the case in the development of many commercial applications. In such a situation one hardly has any control over the design of the building components. Although information regarding the performance of these components can be obtained from measurements performed on the components in isolation, the most important aspect that needs to be controlled is what the resource consumption is for the combined execution of these components at any point in time (performance composition analysis).

The present article will introduce our approach in tackling this question while concentrating on the CPU as a first step in a larger endeavor to find methods of prediction for multiple resources (ex: CPU, memory, bus).

We conducted our experimental studies by considering streaming applications running on a TriMedia device, which integrates a single VLIW processor. The software architecture used for developing the streaming applications chosen was the *TriMedia Streaming Software Architecture*, described later. In this setting, the major source of unpredictability comes from the fact that multiple tasks are executed concurrently. Each time that a task is stopped and another is allocated to the CPU, a Context Switch (CS) occurs. Given the fact that each CS introduces an overhead in terms of processing time (related to cache utilization), being able to predict the Number of Context Switches (NCS) occurring during the execution of the application is critically relevant for predicting the CPU needs for the entire execution. Given the above reasoning, we focused our efforts on developing a method for the calculation of NCS.

A comparable work has been done in the context of the RACA project of the Philips Research Laboratories Eindhoven [5], [6] where an estimation of NCS was provided. The difference between the RACA approach and ours, is that in the context of the RACA project the NCS is estimated based on the estimated number of packets transmitted by each streaming component, and our approach consists of a calculation method for NCS based on a characterization of streaming applications execution.

The article is structured to present first the TriMedia Software Architecture (section 2), which lays out the basics of our streaming model of execution, followed by a characterization of streaming application executions in section 3, from which we deduce our NCS calculation method in section 4. We present an instance of our experimental case studies that validates the method in section 5 and we conclude our presentation in section 6. The last section includes the complete list of documents to which we refer in the text of this article.

II. TRIMEDIA STREAMING SOFTWARE ARCHITECTURE

The TriMedia Streaming Software Architecture (TSSA)[7], provides a framework for the development of real time audio-video streaming applications executing on a TriMedia chip. In general, a media processing application can be described by means of a graph in which the nodes are software components that process a data stream, and the edges are finite buffers (queues) that transport the data stream from one component to the next component in the graph. Data travels in packets between components and the packets are constructed according to a format specified by the application. The TSSA framework provides an Application Programmer Interface (API), which allows constructing and connecting components, as well as the accepted formats for the data stream to be processed and transmitted.

The execution of the components is carried out concurrently and is controlled by the streaming application that instantiated the components (Figure 1). Typically, following the instantiation, the application starts the components, after which it enters a loop during which the components involved in the streaming process

carry out their execution concurrently. According to this scenario the components continue their execution as long as there is input available, or as long as the condition for ending the loop (for instance a stop command from the user) is not fulfilled.

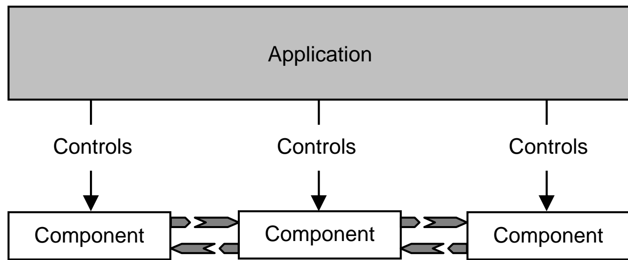

Figure 1. Streaming application controlling the execution of the instantiated components.

The concurrent execution of the components is accomplished by assigning a fixed priority to each task associated with a component. At any given moment in time the system will execute the task with the highest priority that has enough input to run (is not blocked).

Another important aspect of the TSSA streaming components is the memory recycling mechanism (Figure 2). According to this mechanism every connection between the output of a component and the input of another is implemented by means of two data queues. One queue carries *full* packets containing the data to be sent from one component to the next (called Full Queue), while the second 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.

A typical *execution scenario* of a TSSA component (Figure 2) prescribes that the component first gets n full packets from the input Full Queue ❶, then gets 1 empty packet from the input Empty Queue ❷, performs the processing ❸, after which it will recycle the n input packets by putting them in the output Empty Queue❹. The last step is to use the packet received from the input Empty Queue to store data to be transmitted to the next component. The packet will be put in the output Full Queue of the component ❺. Steps 2, 3 and 5 are repeated in this order for m times meaning that after getting n full input packets and m empty input packets the component will have produced m output full packets and n output empty packets. The n *to* m relationship described above is specific to each component.

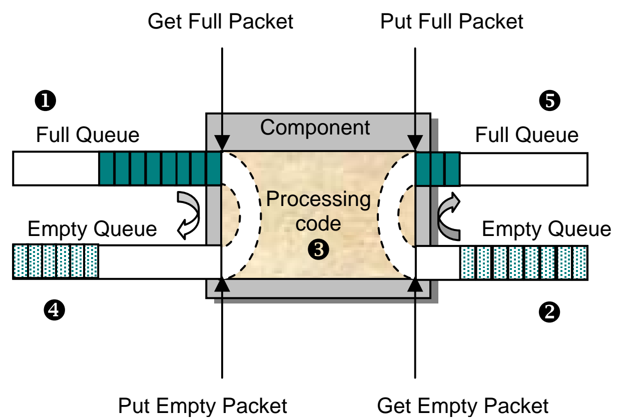

Figure 2. A basic streaming component [7].

The important implication of this type of data transmission management is the fact that any component that has enough input full packets to execute cannot run unless it also has enough empty packets (at least one) in its input empty queue. As we will see in the next section, this fact is highly relevant in calculating the NCS.

III. A CHARACTERIZATION OF STREAMING APPLICATIONS EXECUTION

One of the first questions that comes to mind when attempting to find an calculation method for the NCS is what are the causes of a context switch? As we emphasized above, the execution of the tasks on which the TSSA components map is concurrent, and the decision regarding which task will execute at a particular point in time is determined by:

- the priority of the task and
- the availability of the required number of full and empty packets in the input queues of the component.

During the execution of a streaming application each of the tasks involved will be in one of the following three states: a *blocked state* if the task cannot execute due to lack of input, a *ready-to-run state* when the task could execute (has enough input) but it does not because there exists a task with a higher priority that runs at the moment, and the *running state*. Context switches occur due to blocking, preemption, and due to task execution end.

A second aspect that plays a significant role in finding a method for the calculation of NCS is the nature of this execution. In the present article we will focus on the case of streaming applications consisting of a single linear streaming chain (Figure 3) while later work will extend to multiple chains composing a full graph.

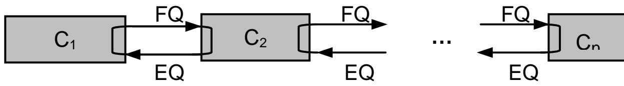

Figure 3. Application consisting of a single streaming chain.

We based our NCS calculation method on the observation that running streaming applications, after an *initialization phase* adopt a pattern of execution that repeats after a specific interval of time. We call this interval *hyperperiod* and the execution of the application according to a repetitive pattern the *stable phase*. The execution of the application ends with a *finalization phase* during which the last transactions in the queues components are completed and the components are stopped. Although we will not provide a generalized proof for the above-described phenomenon, we will present the explanation for the case presented below. For simplicity we will consider that the priorities are assigned from left to right in a descending order as shown in Figure 4, and that all components consume 1 input full packet, and one input empty packet in order to produce 1 output full packet and 1 output empty packet.

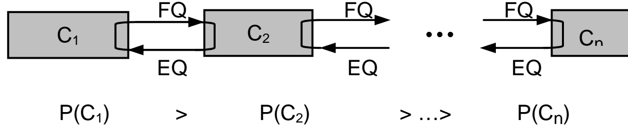

Figure 4. Priorities assigned to components in descending order.

In the case of a chain composed of *n* components, in the beginning of the execution of the application all full packet queues are empty and all empty queues are filled up. Component C_1 makes the link between the application and the source that provides the stream. As an example C_1 can be a component that reads the stream from a storage facility. C_1 is connected to the neighbor component only by two queues and as such it can be blocked if the FQ is filled or EQ is drained. C_n is the component that outputs the processed stream. Usually the output of C_n is sent to a video rendering device such as a TV screen or a computer monitor. C_n is also connected only by two queues to its neighbor which means that it can be blocked is its input FQ is empty or output EQ is filled.

The initialization phase of the streaming application begins with the execution of C_1. Given the fact that C_1 has the highest priority in the chain it will run until it fills up its output full queue (and drains its input empty queue) when it becomes *blocked*. C_2, which has the next highest priority in the chain, will take over and execute until it releases the empty packet, which de-blocks $C_1$❶. C_1 has a higher priority thus it executes again, produces 1 full packet which fills up the output full queue and becomes blocked once again❷. Steps 1 and 2 repeat until the output full queue of C_2 is filled and C_2 becomes *blocked.*

At this moment *C_1 and C_2 are blocked* and the only component *ready-to-run* is C_3. C_3 will take over and execute until it releases the empty packet, which *de-blocks* $C_2$❸. C_2 becomes the highest priority *ready-to-run* component in the chain, so it will execute until it releases the empty packet, which de-blocks $C_1$❹. As explained at step 2 C_1 has a higher priority thus it executes again, produces again 1 full packet, which fills up the output full queue and becomes blocked❺. Steps 3 to 5 will repeat until the output full queue of C_3 is filled and C_3 becomes *blocked.* A more concise representation of the execution presented above is illustrated in Figure 5.

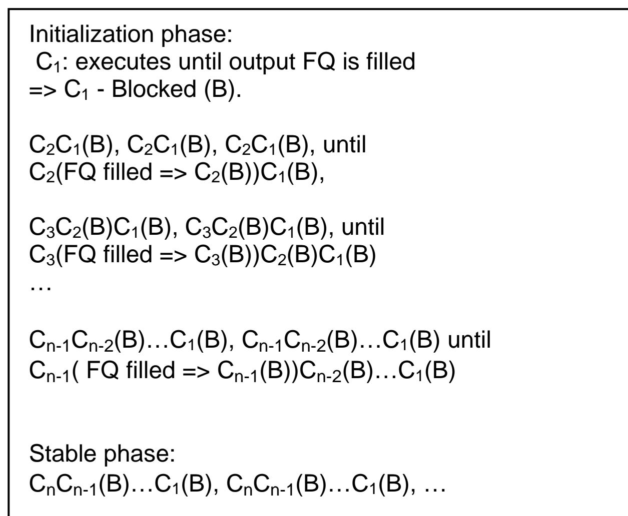

Figure 5. Execution sequence during initialization and stable phases.

The execution of the components continues in a similar fashion with the rest of the components in the chain until all components C_1,…C_{n-1} are *blocked* and C_{n-1} depends on C_n to consume a full packet *and* deliver an empty packet in order be able to resume its execution.

This type of dependency is propagated down the chain with C_{n-2} being dependent in the same way on C_{n-1}, C_{n-3} dependent on C_{n-2}, to C_1, being dependent on C_2.

After this moment, the execution of the chain adopts the following sequence: $C_nC_{n-1}…C_1$, $C_nC_{n-1}…C_1$, C_n… As we can observe the execution of the chain reaches the stable state where it adopted the repetitive pattern $C_nC_{n-1}…C_1$. The stable state lasts until the end of the stream, that is until C_1 can still produce data to be transmitted to the subsequent components. After this moment the components effectuate the last transactions in the queues after which they are stopped.

A few important observations to make from the above example are the following:

- When all queues are either filled or drained most components will be dependent on others (to consume or respectively produce packets) in order to resume their execution. We call the components that de-block others by providing them with the necessary packets to resume their execution *driving* components.
- A driving component can be also dependent on another component. Ex: in the case described above (Figure 4) C_2 is a driving component for C_1 but it also dependent on C_3. The only component in the chain that is driving and not also dependent is C_n.
- At the end of the initialization phase or beginning of the stable phase (when all queues are either filled or drained), the dependent components will be blocked and the driving components that are not dependent, will be ready-to-run. The driving component with the highest priority will run.

The importance of establishing the dependencies between the components at the beginning of the stable state has the purpose of identifying the states in which all tasks are at that moment. Once that is known, finding out the repetitive pattern of execution during the stable state (and as a consequence the NCS during a hyperperiod) is only a matter of applying the scheduling algorithm of the operating system.

Having presented the most relevant information regarding the phases characterizing the execution of streaming applications we are ready to proceed with explaining the NCS calculation method itself.

IV. NCS CALCULATION METHOD

The NCS Calculation Method is based on the property of repeatability according to a pattern that characterizes the execution of streaming applications. This property simplifies significantly the complexity of the task of calculating the NCS because it means that if we could obtain the NCS occurring during one hyperperiod, then we can deduce the entire NCS occurring during the stable phase. By approximating the execution of the streaming application with its execution during the stable phase we obtain an approximation for the NCS occurring during the entire execution of the application.

Step 1. Establish component models. The component models must specify information that characterizes the component:

- The priority assigned to the task associated with the component,
- The execution scenario of the component,
- The average computation time (CT) of the component when executing in isolation,
- The average computation time necessary for the production of each full or empty packet when executing in isolation,
- The relation between the number *n* of input packets necessary in order to produce *m* output packets – again an average.

Step 2. Determine the states of all component tasks involved at the beginning of the stable phase.

Step 3. Apply the following algorithm for identifying the NCS during a hyperperiod of the streaming application execution.

a. Initially consider NCS_hyperperiod = 0.
b. Consider the three states in which a component task can be during the execution of an application: blocked, ready to run, or running. In the beginning of the stable state all dependent components will be blocked while the driving components that are not dependent will be ready to run.
c. Amongst the ready components the one with the highest priority will be running.
d. Follow the execution scenario of the running component specified in its model. After the production of each packet – empty or full check whether delivering the packet de-blocks one of the neighbor components.
   - If yes, that component will become ready to run. Check if that component has a higher priority than that of the currently running component. If yes, the currently running component will be preempted and NCS must be incremented. The algorithm resumes from point d.
   - In not, the currently running component will continue its execution and the algorithm resumes from point d.
e. The algorithm ends when the initial situation observed at the beginning of the algorithm is repeated – the driving component will execute again the same sequence as in the beginning. This is the end of the hyperperiod.

In order to verify the correctness of the calculation we will use the following steps:

Step 4. Determine through measurement the Length of the Application Execution (LAE). LAE is the total duration of the streaming application execution from the beginning of the initialization phase to the end of the finalization phase.

Step 5. Determine the number of hyperperiods (NH) fitting in LAE:

$NH = \lceil LAE/HL \rceil$, where HL is the hyperperiod length determined at the Step 3.

Step 6. Determine the total calculated NCS when approximating the entire execution of the application with the stable state:

NCS_total = NH * NCS_hyperperiod.

Step 7. Compare measured NCS with NCS_total obtained at Step 6.

Without including formulas, we will mention that in this manner we can obtain not only the NCS occurring during a hyperperiod but also more detailed information regarding the NCS due to blocking, preemption or normal end of execution. Additionally, given the fact that the average times needed for producing a full or empty packet are known from the model of the component we can also give an account about the activation times of the component tasks that are resumed as a result of the production/consumption of a packet. Finally, in the same fashion, we can calculate the response times of all the components tasks executing during a hyperperiod.

V. EXPERIMENTAL VALIDATION

In the following we will present one of the experiments we performed in order to validate the method explained previously. The case study we chose to describe here consists of a streaming chain extracted from a larger DVD player application. The components involved are the following (Figure 6):

- A file reader (FRead), which reads information from the disk and converts it into full packets having the format specified by the application. The packets are transmitted to the next component in the chain,
- A video decoder (VDec) that receives the full packets from FRead, decodes them, and transmits the decoded information as packets to the next component in the chain,
- A sharpness enhancement (SSE) component that processes the packets received from VDec such that the final information displayed on the TV screen has an appropriate contrast,
- A video renderer (VO) that displays the information received from SSE on the screen of the TV and is activated periodically. VO is a component with a periodic execution.

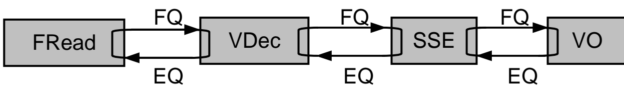

Figure 6. Streaming chain performing video decoding, sharpness enhancement and display on TV screen of information read from storage disk.

In the following we will present how we applied the NCS calculation method for the case study shown above. We will present the most relevant reference points that lead to our results. As such:

Step 1. We identified the values of all entities mentioned in the previous section for this step. Important to mention is the priority assignment of the tasks associated with the components in the chain. In descending order of priorities: FRead, VDec, SSE and VO.

Step 2. When considering priorities, we observe that FRead during the initialization phase has the opportunity to fill the FQ to VDec(the EQ becomes empty) before VDec will have the opportunity to consume packets. As such, after filling up the FQ to VDec, FRead becomes dependent on VDec to consume from FQ and to place one empty packet (EP) in the EQ to FRead. Which implies that FRead becomes blocked and depends on VDec to de-block it. A similar reasoning can be used for all the other components due to the priority assignment to the component tasks. The situation after all full packet queues have been filled has been illustrated in Figure 7 (arrows indicate dependencies between components, ex: FRead depends on VDec to de-block it when the FQ is full, etc.).

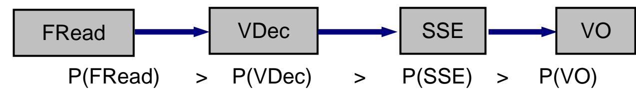

Figure 7. Dependencies between components when considering priorities.

The reasoning above implies that at the beginning of the stable state all components but VO will be blocked depending on the execution of VO to de-block SSE which will de-block VDec, etc. According to the same reasoning, at the beginning of the stable state VO will be running.

Step 3. Apply algorithm for identifying the NCS during one hyperperiod and the hyperperiod length:
HL = 130.4 ms,
NCS_hyperperiod(FRead) = 5,
NCS_hyperperiod(VDec) = 9,
NCS_hyperperiod(SSE) = 8,
NCS_hyperperiod(Vo) = 8.
=> the total NCS_hyperperiod = 5+9+8+8 =30;

Step 4. LAE = 5052.17 ms

Step 5. NH = $\lceil$LAE/HL$\rceil$ = 39.

Step 6. NCS_total = 39*30 = 1170;

Step7. NCS_totalMeasured = 1197;

The difference that we notice between the calculated value and the measured one comes from the fact that we approximated the entire execution of the application with the stable state and because in our component models we used averages for the processing rates of all components. A further step of making the method even more exact

would be to consider distributions of processing rates relative to time.

VI. CONCLUSIONS

In conclusion, our article introduced an approach to tackling the predictability challenge that characterizes real-time embedded systems. The approach concentrates on predicting the CPU needs of a streaming application, as a first step in a larger endeavor to find methods of prediction for multiple resources (ex: CPU, memory, bus). As such, we presented a method for calculating the NCS occurring during the execution of the streaming application, which allows us to predict the overhead introduced by the context switches induced by the combined execution of the components that make up the application. The calculation method is based on the property of streaming applications execution to adopt a repetitive pattern after a short initialization phase. As a result of determining the NCS during the repetitive pattern, we can deduce the NCS during the stable phase. By approximating the execution of the streaming application with its execution during the stable phase we obtain an approximation for the NCS occurring during the entire execution of the application.

REFERENCES

[1] B.Thomas, L.Jo. Performance characterization and modeling. Philips Research Technical Note PR-TN-2004/0000.
[2] E. Eskenazi, A. Fiukov, D.K. Hammer. Performance prediction for software architectures. *Proceedings of the 3rd PROGRESS workshop on embedded systems.*
[3] C. Smith and L. Williams. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, 2001.
[4] F. Aquilani, S. Balsamo, P. Inverardi. An Approach to Performance Evaluation of Software Architectures. Research Report, CS-2000-3, Dipartimento di Informatica Universita Ca' Foscari di Venezia, Italy, March 2000.
[5] D.J.C. Lowet. Performance composition in TSSA. Philips Research Technical Note, PR-TN-2003/00255
[6] D.J.C Lowet. RACA Literature survey: Lierature survey about resource aware component based design. Natlab Technical Note 2002/240
[7] Philips TriMedia Documentation Set. SDE, version2.1