AdaptStream: Towards Achieving Fluidity in Adaptive Stream-Based Systems

Yu Liu
Lero@TCD
School of Computer Science and Statistics
Trinity College Dublin
Dublin 2, Ireland
yuliu@cs.tcd.ie

René Meier
Lero@TCD
School of Computer Science and Statistics
Trinity College Dublin
Dublin 2, Ireland
Rene.Meier@cs.tcd.ie

ABSTRACT

Stream-based systems are frequently subject to changes in their operational environments due to fluctuations in the available computation and communication resources. Dynamic adaptation is a mechanism to improve the fitness of such systems. However, adaptation can block one or more streams thus inadvertently affecting the timeliness properties of streams. This paper describes AdaptStream, an adaptation framework that provides timeliness support for stream-based adaptations. We introduce the concept of fluidity to measure the temporal alignment of stream synchronization during adaptation. We present a scheduling algorithm that calculates the time-bounded schedule of adaptation actions on multiple streams to achieve the fluidity requirement that is traded off against available resources and the smoothness requirement of individual streams.

Categories and Subject Descriptors

D.11 [**Software/Software Engineering**]: Software Architectures

General Terms

Algorithms

Keywords

Dynamic Software Adaptation, Timeliness, Inter-Stream Synchronization

1. INTRODUCTION

Stream-based systems, such as vehicular multimedia content sharing, are deployed and required to function continuously in dynamic environments. Dynamic adaptation at runtime improves the fitness of a stream-based system to the variations in its operational conditions, such as failure of their system parts, resource fluctuation, and changing user requirements. However, adaptation usually entails a period when the quality of the data streams in a system is adversely affected.

Dynamic adaptation can block one or more streams, thus causing delay in the delivery of stream data and driving streams out of synchronization with each other. There are two major synchronization techniques [1] in stream-based systems to maintain the quality of streams: intra-stream synchronization maintains the temporal relationship within a single time-dependent stream; inter-stream synchronization maintains the temporal relationships between multiple streams, such as lip-synchronization of audio and video [9]. The concept of *fluidity* proposed in this paper characterizes the impact of dynamic adaptation on the quality of data streams, and fluidity is a measure of the extent to which temporal alignment of data streams is met during the adaptation period. To achieve fluidity, not only should the impact of adaptation on intra-stream synchronization be considered, but also the impact on inter-stream synchronization. Existing approaches to address the impact of dynamic adaptation can be divided into two groups. The first group [2][3][4] deals with the steady-state conditions of dynamic adaptation, and does not explicitly address the quality of streams. The second group focuses on the quality of intra-stream synchronization during dynamic adaptation [5][6][7]. However, both groups of approaches fall short in addressing the requirement of fluidity in real-world stream-based systems.

In this paper, we present AdaptStream, a novel adaptation framework, designed to support the construction of dynamic adaptive stream-based applications. The framework maintains the explicit separation between the model layer and the component layer of an application at runtime. The runtime model allows us to specify the high-level timing constraints of streams. It carries out checks for the steady-state conditions of adaptations and supports admission control to allocate resources to streams. The component layer consists of running components that process live streams. Once dynamic adaptation has been committed in the runtime model, a reconfiguration scheduler computes and executes a schedule of the sequence of adaptation actions in the component layer. This paper focuses on the mechanism to achieve the fluidity of stream-based applications during adaptation.

The novel contributions of this paper are: 1) it proposes the concept of fluidity to measure the impact of adaptation on the quality of data streams; 2) it presents a time-bounded scheduling algorithm to address the fluidity requirement.

The remainder of the paper is structured as follows: Section 2 introduces real-world stream-based adaptation scenarios and discusses their fluidity requirement; Section 3 introduces the system model of AdaptStream; Section 4 illustrates the adaptation mechanism of AdaptStream. Our algorithm for achieving the fluidity requirement in stream-based systems is presented in Section 5. Section 6 evaluates our approach. Section 7 discusses related work and Section 8 concludes this paper and identifies future work.

2. ADAPTATION SCENARIOS

To effectively track and manage the quality of streams, a path, which consists of a sequence of connected components, is used as the primary application-level construct for stream-based adaptations. Real-world stream-based applications may contain many paths, the relationships between these paths can be complex, and adaptation usually affects more than one path. The following scenarios illustrate the main challenges associated with stream-based adaptations.

Temporal Constraints. Lip-synchronization [9] is the canonical example of inter-stream synchronization. During the play-out process, the audio stream of a speaker's voice and the video stream of the movement of a speaker's lips should match. Adaptations on both streams, such as replacement of encoder and decoder to improve the quality of media content, may introduce delay to the delivery of stream data and then drive the streams out of synchronization. To achieve a glitch-free viewing experience, adaptations on both paths should be appropriately scheduled to ensure that they complete at the same time.

Resource and Temporal Constraints. In a television studio, video streams related to a program are selectively mixed with each other [10]. For instance, in weather forecasting, there are two streams: the stream showing the geographic locations annotated with weather information and the stream showing a weather person standing in front of a wall. It is essential to maintain the temporal alignment of the media data in both streams during adaptation. In line with the resource constraints of the production environment, we have to schedule the distribution of the resources amongst the streams that are being adapted.

The smoothness condition [8] has been proposed to achieve the glitch-free adaptation of a single stream, however, the smoothness condition is inadequate for addressing adaptations involving multiple streams. Therefore, we propose a more comprehensive measure of adaptation in Definition 1 to consider the synchronization relations across streams.

Definition 1. Fluidity is defined as the degree of the temporal deviation between a pair of streams introduced by dynamic adaptation, captured as the percentage of data units that are out of synchronization with each other. The following function calculates the aggregated fluidity from the pairwise deviations between streams. Δ_i is the fluidity of the ith pair of streams and ω_i is the weight associated with this pair. The weight values are set based on the priorities of different pairs of streams.

$$\textit{fluidity} = \sum_{i=1}^{n} \omega_i * \Delta_i, \text{ where } \sum_{i=1}^{n} \omega_i = 1.$$

3. SYSTEM MODEL

The runtime model in AdaptStream expands on the reconfigurable data flow (RDF) system model [12]. The RDF model supports the general semantics of stream-based applications. The RDF model consists of two main entities: components and connectors. A component is an entity that consumes data from input ports, processes the data and sends out the result to its output ports. A connector is a shared space for the communication between two components.

Definition 2. A *path* in a stream-based system is defined as a pair $P = <Comps, Conns>$ where *Comps* is the set of components $\{a_i | 0 \leq i \leq n\}$ and *Conns* is the set of relations between components. The following predicate holds: $\forall i(i \in [0, n) \rightarrow \exists c(c \in Conns \wedge c = a_i \times a_{i+1}))$. a_0 is the source and a_n the sink of the path.

Definition 3. A *path-map* contains a set of paths $PM = \{P | isPath(P)\}$. The following predicate holds for PM: $\exists x(x \in PM \wedge isMaster(x) \wedge \forall y(y \in PM \wedge (isMaster(y) \rightarrow x = y)))$. Function *isPath* returns true if P is a path; function *isMaster* returns true if P is a master path.

Definition 2 gives the formal definition of a path. A path consists of a sequence of ports and components through which a stream flows. Users specify the end-to-end properties, such as latency or jitter, for a path. These end-to-end properties serve as the high-level requirements for adaptation admission and scheduling. Definition 3 states that a stream-based application is characterized by a set of related paths where there is only one master path. The master-slave relationship has been effectively explored by many inter-stream synchronization techniques [13] in stream-based systems. The play-out point in the stream flowing through the master path is taken as the reference for the temporal alignment of other streams.

AdaptStream supports four types of adaptation actions, which are applied to components. These actions are the building blocks of more complex real-world adaptation scenarios and are the basic units of adaptation management. The execution of an adaptation action requires changes on both the runtime model and the stream-processing components.

- *addComponent(comp)*: Creates a new component and then schedules it for execution.
- *removeComponent(comp)*: Removes a component.
- *replaceComponent(oldComp, newComp)*: Replaces a component, which might also lead to an interface change, such as add a new port or modify the data type of a port.
- *upgradeComponent(oldComp, newComp)*: Replaces a component, however the interface of the component remains unchanged.

4. ADAPTSTREAM

Adaptations of stream-based applications have unique set of requirements, which are addressed in the AdaptStream framework. The most important requirements relevant to the scope of this paper are as follows:

Consistency. Adaptation should transform a system from one consistent configuration to another. With respect to data streams, consistency means that adaptation does not cause any data to be mishandled, corrupted or lost.

Timeliness. Adaptation on a path does not violate the smoothness property of a data stream, meaning that data units arrive at a near-constant rate without noticeable jit-

ter. If multiple streams with a synchronization relationship are involved in an adaptation, then the synchronization relationship should still be maintained during adaptation.

4.1 Steady-State Conditions

An atomic step comprises a set of adaptation actions, and either all these actions are executed to completion or none of them are executed. Execution of an atomic step leads a system from one steady state to another. Consider the scenario where the media content is adapted from MPEG-1 to H.263. This scenario illustrates an atomic step that consists of two actions: 1) replace an MPEG-1 encoder component with an H.263 encoder component; 2) replace an MPEG-1 decoder component with an H.263 decoder component.

Definition 4. An *atomic step* of a path P is defined as $AS \subseteq P.Comps$. The following predicate holds for AS: $\forall x(x \in AS \rightarrow \forall y(depends(x,y) \rightarrow y \in AS))$. The function *depends(x,y)* returns true if component x is dependent on component y.

The concept of atomic step is built upon the dependency relationships amongst components and it strengthens the consistency of an application. Dependency relationships derive from application semantics, the richer the semantic model the more complex dependencies can be specified [14]. We assume that such information is provided by the user as part of the adaptation requirements. The order of execution of adaptation actions within an atomic step is from the most upstream components to the most downstream components, as data streams are always flowing from the source to the sink of a path. An atomic step is considered as a whole for resource allocation. Either the desired amount of resources is allocated for an atomic step to execute or no resources are allocated.

4.2 Admission Control

The purpose of admission control is to reserve resources for the new components as well as for the execution of adaptation actions, and admission control should not have observable effect on the running streams. The estimation of resource consumption is carried out using techniques proposed in [15]. AdaptStream adopts the worse case estimation to accommodate the peak resource demand during this stage. For instance, to provide optimum picture quality, each video encoder maintains a local buffer with a size comparable to that of the data buffer at the decoder's end. The worse case situation is to allocate the largest possible buffer for the encoder to prevent overflowing. Our algorithm for determining the worst-case resource consumption traverses all components and sums up their worse-case resource needs.

4.3 Live Switch

A segment is defined as a sequence of components to be adapted between two fixed boundaries. Live switch executes after admission control and is triggered by an event indicating the readiness to change a live stream. Such events can be a user clicking a button through an interactive program, certain points in a data stream being reached or a control event being released by another component. Live switch transforms the running configuration from the old segment to the new segment. As shown in Figure 1, the H.263 encoding segment is replaced with the MPEG-1 encoding segment. The boundaries of each segment are clearly indicated by the rectangle covering it. The output port of the video source is first disconnected from the input port of the H.263 Encoder component, and then connected to the MPEG-1 Encoder component; similarly, the input port of video sink is switched from the H.263 Decoder component to the MPEG-1 Decoder component.

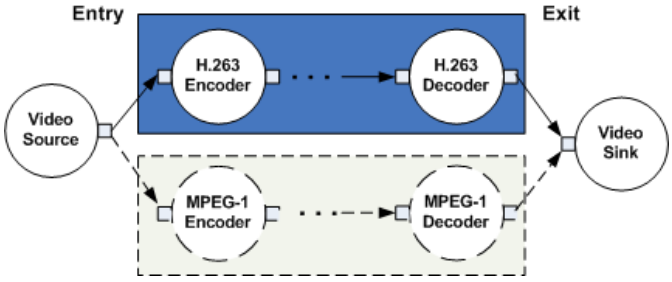

Figure 1: Live switch from the old segment to the new segment

5. ACHIEVING FLUIDITY

The adaptation solution employing intra-stream synchronization only achieves the smoothness condition [6]. To accommodate the fluidity requirements of multiple stream adaptation, we first introduce the conditions for fluidity-based scheduling, then we present our scheduling algorithm and discuss its trade-offs in terms of timeliness and resource usage. Our discussion focuses on the synchronization between one master stream and one slave stream.

5.1 Latency Guaranteed Region

There are two main challenges facing the problem of achieving fluidity for multiple streams. Firstly, the style of synchronization is solution specific: techniques in multimedia synchronization [1] use buffers to smoothen the network jitter at media source, media receiver or intermediate components. We cannot assume a particular style of inter-stream synchronization while developing the generic scheduling algorithm to counteract the side-effect of adaptation. Secondly, adaptation on each path should be associated with a time bound so the asynchronization between data streams can be effectively handled by our scheduling algorithm.

We propose the notion of *Latency Guaranteed Region* (LGR) to cope with the above challenges. An LGR is a sequence of components and ports between two fixed points in a path. The latency of an LGR is guaranteed under normal operational conditions and the jitter of an LGR is the time bound for the execution of any adaptation actions within this region. On the one hand, the boundaries of an LGR naturally serve as the synchronization points. An LGR can represent the entire path so that synchronization is carried out at the source and the sink or it can represent only part of a path. On the other hand, an LGR can guarantee the worse-case latency and only those adaptation actions that fit in this worse-case bound are admitted by admission control.

5.2 Parameters of Schedule

The following set of parameters are important for the calculation of a schedule that achieves the temporal alignment between a master stream and a slave stream.

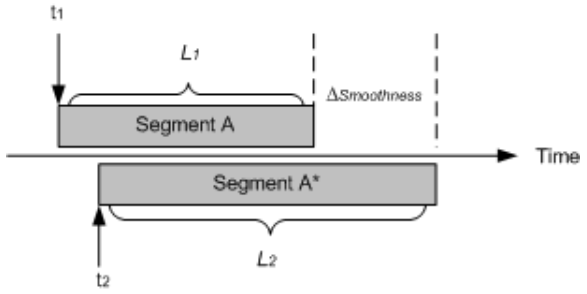

Figure 2: Smoothness condition for single stream adaptation

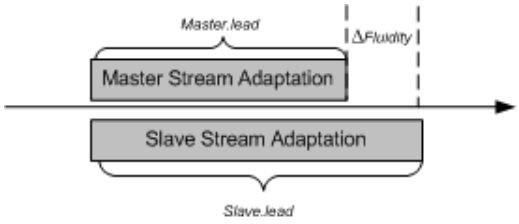

Figure 3: Adaptations on two streams are synchronized to achieve fluidity

• *Segment Latency.* L is given by the latency of a segment. A segment has a higher latency when another segment is running alongside it (i.e. during live switch).

• *New Segment Startup.* t_1 is the point in time to connect the entry of a new segment into a running configuration to allow the upstream data to flow through.

• *Old Segment Terminate.* t_2 is the point in time to disconnect the entry of an old segment from the running configuration.

• *Lead Time.* t_{lead} is the elapsed time between requesting a segment switch and its completion [6].

• *Smoothness.* $\Delta_{smoothness}$ represents the degradation of quality of a stream during adaptation. It is calculated as the interval between two streams when they are spliced together.

• *Fluidity.* $\Delta_{fluidity}$ represents the degree of temporal alignment between a master stream and a slave stream.

Figure 2 illustrates how to measure the smoothness of an adaptation taking place on a single stream. At time t_1, the entry of segment A is disconnected from its upstream neighbor, while the exit of the segment remains connected to its downstream neighbor until all data that has entered is processed and flushed out. Time t_1+L_1 signals the end of the old stream, as segment A stops producing new data and is therefore removed from the current configuration. Similarly, the entry of segment A* is connected at time t_2 and the exit of the segment is connected at time t_2+L_2 when the new stream appears. A smoothness algorithm [6] schedules the execution of adaptation actions in such a way that $\Delta_{smoothness}$ is minimized.

In Figure 3, the fluidity property $\Delta_{fluidity}$ expresses the requirement for adaptation of one stream to finish at the same time as the adaptation of the other stream. The temporal alignment of two streams still holds if the delay of either stream induced by adaptation can be compensated by an appropriate scheduling algorithm. t_{lead} of the master stream may be longer or shorter than t_{lead} of the slave stream. The fluidity scheduling accommodates both situations by injecting trigger events of a segment switch at different points in time. The smoothness scheduling is nested in the fluidity scheduling, as the former addresses the glitch-free switch on a single stream and the latter coordinates the temporal alignment of multiple streams.

5.3 Scheduling Algorithm

We propose a fluidity scheduling algorithm in this section. The goal of the algorithm is to achieve a reasonable fluidity value for the inter-stream adaptations under different resource conditions. It is required to function under both optimistic conditions and pessimistic conditions. Optimistic scheduling caters for the average-case reservation of resources and for applications that have a moderate degree of out-of-sync tolerance. Pessimistic scheduling is suited for applications with stringent synchronization requirements, and is based on worse-case reservation of resources. Our scheduling algorithm consists of three phases: 1) Partition; 2) Compensation; 3) Integration.

Partition. During this phase, adaptation actions in an atomic step are mapped onto the LGRs by means of a mapping function. To ensure consistency, atomic steps of a path have to be executed sequentially rather than in an interleaved fashion. The partition phase carves up an atomic step into an ordered set of segments, and groups those segments that fit into one LGR together. The remainder of our algorithm addresses the temporal alignment of two LGRs, one is from the master stream and another is from the slave stream: $\text{LGR}_{master} = \{\text{sg}_i |\ 0 \leq \text{i} \leq \text{n}\}$, $\text{LGR}_{slave} = \{\text{sg}_i |\ 0 \leq \text{i} \leq \text{m}\}$.

Compensation. To ensure that the adaptation actions on each LGR do not drift data streams out of synchronization with each other, the difference in their lead time has to be compensated. Suppose the lead time of the execution of LGR_{master} is $Lead_m$, and the lead time of LGR_{slave} is $Lead_s$. Our strategy is to begin the integration of one LGR with the shorter lead time L later than that of another LGR.

$$L = |Lead_m \text{ - } Lead_s|$$

$$L_i = \omega_i * L, \text{ where } \omega_i = \frac{Lead_i}{min(Lead_m, Lead_s)}$$

Listing 1. outlines the pseudo code of the compensation and the integration phase. The lead time of an LGR is calculated by adding up the average-case lead time of each segment in the LGR. To fairly distribute the burden of temporal alignment across all the segments in an LGR, each segment sg_i is associated with a compensation slot L_i defined above (line 3-10). The scheduling of a segment then takes into account the compensation slot to postpone the execution of its stream-processing components. As shown in Figure 4, the lead time of segment sg_i is expanded by L_i and the lead time of segment sg_{i+1} is increased by L_{i+1}.

Integration. This phase schedules the components in a segment to process data streams, by means of injecting the trigger events to control the timing of their execution. It is able to work towards the pessimistic mode where the components begin processing data at the earliest possible time and new data is delivered to the exit of the segment before it replaces the old segment. It can also work towards the optimistic mode where the schedule is based on the average-case timing estimation but the risk is deadline overshoot.

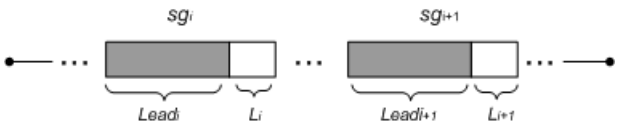

Figure 4: Each segment is compensated by a desired slot in an LGR

```
1  procedure CalculateSlots (LGR1, LGR2)
2   //calculate the compensation slots
3   LGR = min(LGR1, LGR2);
4   minlead = min(LGR1.lead, LGR2.lead);
5   maxlead = max(LGR1.lead, LGR2.lead);
6
7   for (all sgs in the LGR) do
8         sg.slot = (maxlead - minlead) *
9                   sg.lead/minlead;
10  end for;
11 end procedure;
12
13 procedure Integrate (sg, tr)
14 if(Is_admitted(sg, sg.config)) then
15  for (all actors in the sg) do
16        Init actor;
17  end for;
18
19  //stepwise control of adaptation execution
20  while(waitStepTrigger(tr, sg.slot)) do
21          for (all actions in tr) do
22                action.process();
23                action.commit();
24          end for;
25  end while;
26
27  //switch on the new segment sg
28  if(waitSwitch(sg)) do
29          sg.switch(sg.config);
30  end if;
31 end if;
32 end procedure;
```

Listing 1: Scheduling the components to process data streams

An admission test is conducted through *Is_admitted*, which returns true if a segment has the desired timing properties and resource usage in relation to the running configuration. Once the admission test is passed, the components in a segment are initialized (line 15-17): a component is parameterized, activated and connected to its downstream neighbor. However, during the initialization, the components have not yet begun to execute as a result the reserved resources are not utilized. The step-wise control mechanism (line 20-25) paces the progress of stream-processing: a trigger event is received through *waitStepTrigger* and it contains a list of actions to be executed in this step. Stream processing of each component begins by the invocation of *process*, and upon completion, the status is communicated back to the step-wise controller through the invocation of *commit*. The final stage of integration (line 28-30) is to connect the exit of a segment into a running configuration to complete the replacement.

The step-wise controller injects events at proper times to trigger stream processing. The interval between two successive injections should be generous enough to allow the previous step to finish processing, however it also needs to take into account the risk of deadline overshoot. Timing predictive models [16][22] are used to calculate the interval between the injections.

6. EXPERIMENTAL EVALUATION

In our experiment, the AdaptStream framework and the stream-based applications are deployed on an embedded platform Java SunSpots [21], which is a resource-constrained environment for real-time applications written in Java. The goal is to evaluate the effectiveness of our scheduling algorithm under different operational conditions [23], and to show the relations between fluidity and smoothness.

6.1 Experimental Setup

A lip-synchronization scenario is implemented using AdaptStream. The master path carries the audio stream and the slave path carries the video stream. In our experiment, there are 6 components in each path and the region between the sender and the receiver forms one LGR. Two atomic steps are executed periodically on each path: 1) replace a pair of media encoder and decoder; 2) replace a pair of error correction encoder and decoder. We compare the optimistic (OP) and pessimistic (PE) scheduling against the baseline (BL) scheduling. The baseline scheduling employs the smoothness control [6] on every stream and does not support fluidity.

The boundary between the optimistic mode and the pessimistic mode can be adjusted in a continuous range. In this experiment, we choose the pessimistic mode to be 50% of the compensation slot, and the optimistic mode 90% of the compensation slot. The following metrics are sampled during *the period when atomic steps are executed.*

• *smoothness.* Sample the interval between packets at the receiver end and calculate the percentage of deviation from the normal interval value.

• *fluidity.* Sample the output packets from both the master and the slave stream and calculate the percentage of data packets that are out-of-sync with each other (degree of temporal alignment) based on their timestamps.

6.2 Experimental Results

The first result is obtained under abundant resource (memory usage) conditions, we measure the effectiveness of three scheduling algorithms in achieving fluidity with respect to data transmission rate. As is shown in Figure 5, fluidity deteriorates with the increase in the transmission rate of data, revealing its sensitivity to the ongoing communication. When the data rate is relatively low (under 256KB/s), the benefit of pessimistic and optimistic scheduling is only marginal; however when the transmission rate reaches 1MB/s, the number of out-of-sync packets is reduced by nearly 50% by pessimistic schedule in comparision with that achieved by baseline. Pessimistic schedule invariably outperforms optimisstic schedule in each of the five tests (6.7% $\sim$ 19.8%), as pessimistic schedule utilizes more resources and has a lower risk of deadline overshoot. The result shows that the higher the transmission rate the more likely the fluidity property is violated and our fluidity control mechanism performs well.

Then, we measure how effective each scheduling algorithm is in achieving fluidity under varying resource conditions. Data transmission rate is fixed at 256KB/s in this case. Suppose *M1* is the average-case estimation of required memory for executing both the master and the slave stream, *M2*

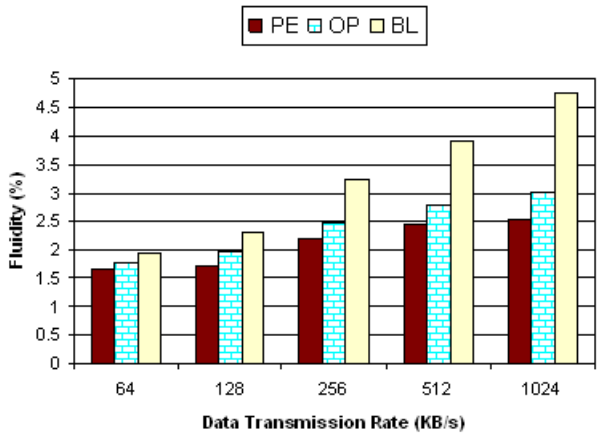

Figure 5: Fluidity under different data transmission rates

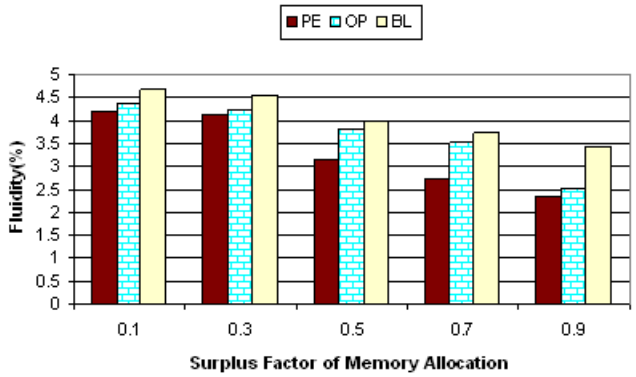

Figure 6: Fluidity under different resource conditions

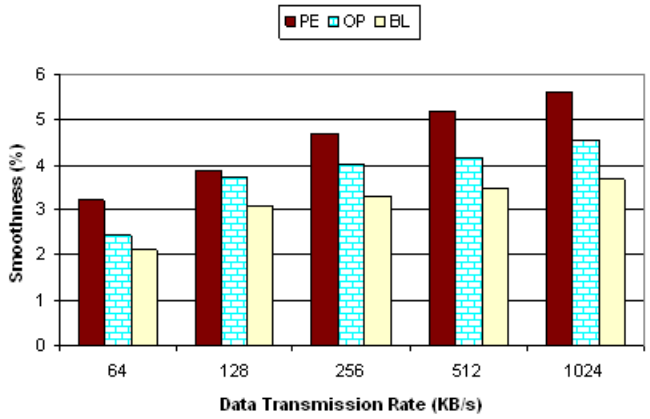

Figure 7: Smoothness under different data transmission rates

is the average-case estimation of the required memory for both streams and their replacement (new segments). The amount resource allocated (x) can be adjusted by either increasing or decreasing the surplus factor $\theta = \frac{x-M1}{M2-M1}$ within the range [0,1]. As is shown in Figure 6, with the increase of the surplus factor, a better fluidity value is achieved by all three scheduling algorithms. The experiment conducted under the abundant resource condition has indicated the lower bound of fluidity: PE 2.2%, OP 2.48% , and BL 3.25%. As θ increases, the value of fluidity is gradually approaching the lower bound: when θ is 0.9, PE achieves 2.34%, OP achieves 2.53% and BL achieves 3.42%. The most fluidity gain is achieved when the value of θ is 0.5 for pessimistic scheduling (32%), and the value of θ is 0.9 for optimistic scheduling (40%). This is possibly due to the different time-bound within the compensation slots we have chosen for each scheduling algorithm.

Figure 7 shows the result of smoothness obtained (master stream) under varying data transmission rates with three scheduling algorithms. Abundant resources have been allocated for each of the five tests. Baseline scheduling achieves better smoothness than the other two algorithms, which is justifiable as baseline scheduling does not deal with the fluidity requirement and as a result is less costly to execute. OP consistently achieves better smoothness than PE (5.4% $\sim$ 32%) and this is attributed to the fact that optimistic scheduling incurs less execution overhead as it has a more generous time-bound.

7. RELATED WORK

Our related work derives from the areas of multimedia synchronization and dynamic software adaptation, as the AdaptStream framework aims at addressing the fluidity requirement of adaptive stream-based applications. Techniques in multimedia synchronization rely upon application-level protocols, such as buffering and re-transmission, to address fluidity requirement, however those techniques do not generally involve software adaptation. On the other hand, approaches to software adaptation provide weak support for the fluidity requirement of stream-based systems.

Multimedia Synchronization. Intra-stream and inter-stream synchronization are the main classes of synchronization control in multimedia systems. *Intra-stream synchronization* maintains the continuity of media units at the output of a single stream [17][18]: it is necessary to avoid buffer underflow and buffer overflow situations at the receiver's end, and the play-out is expected to consume media units at an appropriate rate. *Inter-stream synchronization* preserves the temporal relationships amongst different media streams [19][20]: some streams can be time-dependent such as video and audio while other streams can be time-independent such as static images and text. If the presentation of multiple streams is conducted without inter-stream control, then jitter can gradually build up across streams. Inter-stream synchronization can also be classified into point, real-time continuous and adaptive synchronization [9].

Dynamic Software Adaptation.The first group of approaches focuses on the *steady-state conditions* of dynamic adaptation. Oreizy [2] proposed an adaptation protocol used during the component replacement to achieve a safe adaptation process. Feiler [3] introduced the syntactic and semantic consistency to strengthen the integrity of an adaptive program. Zhang [4] presented the concept of safeness con-

ditions to ensure that adaptation leads an application from one consistent steady-state to another. The second group of approaches only addresses the *smoothness requirement* of a single stream. Hillman [5] presented an approach to manage and measure the cost of reconfiguration in terms of time and disturbance, however it does not allow independent parts of the system to be adapted simultaneously. Mitchell [6] proposed an approach to schedule the updates of a single live multimedia stream to achieve the glitch-free reconfiguration. Zhao [7] introduced influence control of dynamic reconfiguration to avoid data loss and achieve version compatibility. However, as one of the preconditions of his algorithm synchronization between flows is excluded from any configurations.

8. CONCLUSIONS AND FUTURE WORK

This paper has focused on calculating the schedules for adapting multiple streams to maintain end-to-end timing properties within the resource constraints of the underlying platform. The AdaptStream framework is designed to accommodate the steady-state conditions and the timeliness requirements of adaptive stream-based systems. Our main contributions are the models and the scheduling algorithms of the AdaptStream framework for addressing the synchronization requirements of multiple adaptive streams. Experiments have shown that AdaptStream can support inter-stream synchronization under various operational conditions. Future research plans investigate the scheduling algorithms that cater for application specific requirements, such as *syntax-awareness* and *priority-awareness*, and to extend the AdaptStream framework to support these requirements.

ACKNOWLEDGEMENT. This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero, the Irish Software Engineering Research Center (www.lero.ie).

9. REFERENCES

[1] Boronat, F., Lloret, J., Garcia, M.: Multimedia group and inter-stream synchronization techniques - a comparative study. In: Elsevier Information Systems 34 (2009), pp. 108-131

[2] Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on Software engineering(ICSE), p.177-186

[3] Li, J., Feiler, P.H.: Managing inconsistency in reconfigurable systems. In: IEE Proceedings - Software 145(5): 172-179 (1998)

[4] Zhang, J., Cheng, B.H.C., Yang, Z., McKinley, P.K.: Enabling safe dynamic component-based software adaptation. In: Workshop on Architecting Dependable Systems(WADS), pp 194-211 (2004)

[5] Hillman, J., Warren, I.: An Open Framework for Dynamic Reconfiguration. In: Proceedings of the 26th International Conference on Software Engineering(ICSE), pp 594-603.

[6] Mitchell, S., Naguib, H., Coulouris, G., Kindberg, T.: Dynamic Reconfiguring Multimedia Components: A Model-based Approach. In: Proceedings of the Eighth ACM SIGOPS European Workshop (1998)

[7] Li, W., Zhao, Z.: Influence control for dynamic reconfiguration of data flow systems. Journal of Software, 2007.

[8] S.R.Mitchell: Dynamic Configuration of Distributed Multimedia Components. Ph.D. thesis, University of London, 2000.

[9] Chen, T., Graf, H.P., Wang, K.: Lip synchronization using speech-assisted video processing. In: Signal Processing Letters, IEEE (1995)

[10] Bhatt, B., Birks, D., Hermreck, D.: Digital Television Making it Work. In: IEEE Spectrum 34(10), pp 19-28, October 1997.

[11] Blakowski, G., Steinmetz, R.: A media synchronization survey reference model, specification and case studies. In: IEEE J.Sel.Areas Commun. 14(1) (1996)

[12] Zhao, Z., Li, W.: Dynamic Reconfiguration Planning with Influence Control. In: 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)

[13] Manvi, S.S, Venkataram, P.: An agent based synchronization scheme for multimedia applications. In: J.Syst.Software(JSS) 79(5) (2006) 701-713

[14] Feiler, P., Li, J.: Consistency in Dynamic Reconfiguration. In: Proceedings of the 4th International Conference on Configurable Distributed Systems (1998)

[15] Fritsch, S., Clarke, S.: TimeAdapt: timely execution of dynamic software reconfigurations. In: Proceedings of the 5th Middleware doctoral symposium (2008)

[16] Brennan, S., Cahill, V., Clarke, S.: Applying non-constant volatility analysis methods to software timeliness. In: Proceedings of the 21st Euromicro Conference on Real-Time Systems (2009)

[17] Laoutaris, N., Stavrakakis, I.: Intrastream synchronization for continuous media streams: a survey of playout schedulers. In: IEEE Network Mag. 16(3)(2002) 30-40

[18] Ishibashi, Y., Tasaka, S.: A synchronization mechanism for continuous media in multimedia communications. In: Proceedings of the IEEE INFOCOM'95 pp 1010-1019

[19] Qiao, L., Nahrstedt, K.: Lip synchronization within an adaptive VoD. In: SPIE Multimedia Computing and Networking (1997) pp 170 -181

[20] Bourkerche, A., Owens, H.: Media synchronization and QoS packet scheduling algorithm for wireless systems. In: Mobile Networks Appl.10(1-2) (2005)

[21] Java Sun SPOT Application Development using Java ME: http://www.blueboard.com/spot/

[22] Brennan, S., Fritsch, S., Liu, Y., Sterritt, A., Fox, J., Linehan, E., Driver, C., Meier, R., Cahill, V., Harrison, W., Clarke, S.: A Framework for Flexible and Dependable Service-oriented Embedded Systems. In: Architecting Dependable Systems VII (ADS VII), vol. LNCS 6420: Springer-Verlag Berlin Heidelberg, 2010, pp. 123-145, to appear

[23] Liu, Y., Meier, R.: Resource-Aware Contracts for Addressing Feature Interaction in Dynamic Adaptive Systems. In: Proceedings of the Fifth International Conference on Autonomic and Autonomous Systems (ICAS 2009), pp.346-350