Work in Progress: Adaptive Scheduling with Approximate Computing for Audio Graphs

Pierre Donat-Bouillud
Sorbonne Université/STMS/Inria
Paris, France
pierre.donat-bouillud@ircam.fr

Christoph M. Kirsch
University of Salzburg
Salzburg, Austria
ck@cs.uni-salzburg.at

I. Introduction

Interactive Music System (IMS) are highly dynamic programmable authorship systems that combine audio processing and control in real-time in an audio graph. Audio graphs combine several dependent signal processing nodes in an online reconfigurable graph with soft real-time constraints (but more stringent than for video for example).

Composers and musicians mainly use these IMS on mainstream operating systems such as Windows or Linux, where a reliable estimation of the worst execution time (WCET) is difficult, due to complex cache hierarchies, lack of real-time schedulers, lack of temporal isolation between tasks, and the difficulty to predict which tasks are executed at any time.

Hence, we do not assume that we know the WCET of tasks, but we react to the changes in the execution environment, by adapting the execution time of a task with *approximate computing*. We do not only have to degrade one node, but *to choose which nodes to degrade*, while preserving real-time constraints. In the context of signal processing nodes, nodes are considered to be *black boxes* and degradations only occur on the *data streams* between boxes: degradations can be *resampling* or *substituting* a node by another version.

If we consider that time is a resource, in real-time systems, time is often the only resource that is degraded (*i.e.* by missing a deadline). Here, we also degrade other ones, and want to make explicit the tradeoff among various quality metrics of the task, such as the sample rate.

We have started to investigate how nodes to be degraded can be chosen at execution time, *online*. The scheduler must be overhead-aware, and we aim at keeping the computations of the scheduler as low as possible. A model well suited for describing signal processing tasks and the streams between nodes is the *dataflow* paradigm. We will show how we can adapt *approximate programming* to a dataflow graph.

Our contributions are the following ones:

- integrate degradations in the dataflow paradigm,
- how to choose which nodes to degrade online,
- overhead-aware degradations

II. Background and motivations

A. Interactive music systems

IMSs deal with audio streams and are used to perform pieces, described by programs also called *scores*, during a concert in real-time by combining various audio effects. They have to deal with signal processing, with filling audio buffers periodically and sending them to the soundcards, and controls, that can be aperiodic (such as GUI change) or periodic (such as a low frequency oscillator). Puredata [1] is an IMS that graphically shows the audio graph but makes it difficult to change it dynamically. Other IMS are more dynamic, such as ChucK [2]. In Antescofo [3], a computer and a musican interact on stage.

B. Real-time constraints for audio

The soundcard requires audio samples to be written in its input buffer periodically. For example, for the usual CD sampling rate of 44.1 kHz, and a buffer size of 64 samples, the audio period is 1.45 ms. Depending on the targeted latency and the resources of the platform, the buffer size ranges from as little as 32 samples for audio workstations to 2048 samples for some Android phones.

Audio real-time constraints are soft real-time, but the real-time requirements are more stringent than for video. For video streams, dropping a frame does not entail a visible decrease in quality, and thus is used in a lot of video streaming [4] protocols. On the contrary, missing a deadline for an audio task is immediately audible.

a) Underrun: The audio driver uses a ring buffer the size of which is a multiple of the size of the audio soundcard. If the audio task misses a deadline, it does not fill the audio buffer quickly enough. It is called a buffer *underflow*. Depending on the implementation, previous buffers are replayed (so-called "machine gun" effect) or silence is played, leading to cracky audio and clicks due to discontinuities in the audio signal.

b) Overrun: In the same way, filling the audio buffer too quickly will entail audible discontinuities in the output sound.

C. Related work

Some approaches have tackled adaptive scheduling, either by discarding tasks entirely, or by degrading them.

a) Approximate computing: It is a paradigm of computation that allows some errors in computations to improve performance. It relaxes absolute correctness to correctness with a quantified error. The goal is to design systems with a favourable quality vs performance or energy tradeoff. A coarse grain strategy [5] can consist of dividing tasks into a mandatory part and a discardable optional part. This makes it easier to schedule tasks in real-time, with low-overhead, but does not take into account dependencies between tasks. Another computation model [6] uses a graph to represent a programs and generates approximate versions given an error bound. Accuracy-aware transformations are separated into two classes: substitution transformations and sampling transformations. This model does not natively embed time constraints and requires a preliminary phase of profiling, and hence cannot help for dynamic graphs.

b) Resource reservation: A fraction of the CPU processing power is reserved [7] to some tasks. It works well for different competing tasks: for multimedia for instance, video tasks and audio tasks can have various reservations as audio tasks are *more* real-time than video tasks (see Subsection II-B). However, resource reservation does not deal well with identical tasks competing for resources that would require a higher fraction of the CPU.

c) Mixed criticality: In mixed criticality systems [8], the high criticity tasks must be scheduled imperatively, contrary to the low criticality ones, that can be discarded if it could entail missed deadlines for the high criticality ones. In our case though, all the tasks have the same criticality.

III. Model of an audio graph

We want to model an audio graph, for which audio streams flow from input to output at different rates depending on the requirements of the nodes of the graph. The *dataflow model* [9] is well suited to represent such tasks with data dependencies. Here, we do not limit ourselves to *synchronous dataflow* where the rates of the tasks are fixed. However, the *dataflow model* does not take time into account, so we will consider a *timed dataflow* model, where we precise *when* and *how long* a task can be executed.

A. The dataflow model

The *dataflow* model is data-driven. Dataflow graphs are directed graphs where nodes are computations, and arcs are data paths. Data is represented as sequences of *tokens*. The execution is based on the availability of *tokens* on the input of a given node. A node can *fire* when there is enough *tokens* on its input. In the context of audio processing, the audio signal is sampled at periodic intervals. These *samples* correspond to the *tokens* in the dataflow model.

The nodes without inputs ports are called *inputs*: they are typically audio stream generators. The nodes without output ports are the *outputs*: they are audio sinks, to the audio sound card for instance. Node that are neither *inputs* or *outputs* are called *effects*.

B. The timed dataflow model

A dataflow graph does not describe the time instants of firing, but only their partial ordering. However, dataflow graphs are often used to describe real-time processing, for instance, digital signal processing. For the CDs with a 44.1 kHz sampling rate, there is one *sample* every $\frac{1}{44100}$ seconds. In the following, *input* nodes are given firing dates, and all the nodes in the dataflow graph get a current execution time T_e and a mean execution time (MET) M_e. The *output* nodes are given deadlines, which correspond to when the soundcard requires samples.

IV. Quality

The quality of an audio graph is a subjective and relative concept: does this version of the graph sound better or worse than this other one? But comparing degraded and normal versions is not suited for real-time execution. We rather compute an *a priori* quality measure based on some parameters of the computations. The quality should also be a *compositional* measure: the quality of a graph should be computable given the quality of its nodes and edges.

Each node e of the graph is given a quality function q_e.

a) Composing quality: Given two nodes e and e' such that $e \to e'$, if we note t the input data on e and t' the output data of e', we note:

$$q_{e \to e'} = q_e(t, e(t)) \otimes q_{e'}(e(t), t')$$

$\otimes$ is associative but not commutative in general.

For a chain $e_1 \to \cdots \to e_n$, we will assume that $q_{e_1 \to e_2} \leq \min\{q_{e_1}, q_{e_2}\}$, that is to say, the quality never increases on a path.

The quality $q_{\mathcal{G}}$ of graph $\mathcal{G}$ is derived in the same way for all its chains and as the minimum of the quality of every chains that it is composed of.

A. Degrading quality

In a dataflow graph, nodes receive samples and then process them when they have got enough to be fired. Hence, if a node receives samples less often, it will use less processing time. This operation of changing the number of samples is common in digital signal processing, and is called *resampling*: less samples is *downsampling*; more, *oversampling*. To resample, we insert nodes that will change the rate of producing or consuming samples. These resampling nodes are normal nodes: they may interpolate values, copy values between buffers. Hence they also have a quality measure and a MET to take the overhead of this degradation into account.

If we insert a downsampling node, all the nodes that are on a path starting on this nodes will process on a downsampled stream. In case of an output node dictating a specific sample rate, we also have to insert an oversampling node.

B. Measuring the quality

We measure q_e *a priori*: the lower the sample rate, the lower the quality. When audio is output too late, we get a click, a discontinuity (see Subsect. II-B). On the contrary, if the audio stream had been downsampled, there would have been some samples, non-zero values. Thus we assume that a lower sample rate yields a better quality than a discontinuity.

C. *Estimating the error on the whole graph*

We degrade a chain $\mathcal{C}$ of processing nodes $e_1, \ldots, e_n$ by *downsampling*. Let e_{down} and e_{up} the node that respectively downsamples at the beginning of the chain, and upsamples at the end of it. According to the beginning of IV, the error is $q_{\mathcal{C}} = q_{e_{\text{down}} \to \cdots \to e_{\text{up}}}$.

Regarding the processing time of the whole chain, it is at least divided by the downsampling factor r (for instance 2 to downsample from 96 kHz to 48 kHz), as the processing time of a node is no more than linear in the number of input samples: real-time audio programmers always enforce this maximum complexity, and so we assume it here. The whole processing time becomes:

$$T_{e_{\text{down}}} + \frac{1}{r}\sum_{i}^{n} T_{e_i} + T_{e_{\text{up}}}$$

V. Degrading the whole graph

The tasks are dependent tasks and dependencies are given by the audio graph. We can find the schedule with a topological sort on the graph (which means that we assume that the audio graph is acyclic). We estimate the expected remaining execution time before executing each node of the graph by using the MET of each node. Before executing each node, we check whether the expected remaining time is smaller than the actual remaining time, with the deadlines given by the *outputs*. After executing a node, its MET is updated.

We have started to investigate how to degrade the graph online, for a dynamic real-time audio graph. Finding the best tradeoff between quality and lateness is an optimization problem, it may be too costly to solve optimally for real-time systems.

a) Transient overload: A first observation is that given a chain of the same processing nodes, it is better to degrade the nodes at the end of the chain than the nodes at the beginning, due to the property that quality never increases on a chain (see Sect. IV). Another heuristic is to try to minimize the number of resampling nodes we add, while maximizing the number of nodes that are degraded, in order to minimize the degradation overhead. It means that the number of branches to be degraded should be minimized, and thus we explore the graph one branch at a time.

The algorithm works as follows: check before executing every node if there is enough time before the deadline. If not, look for nodes to degrade among the ones that have not been executed, starting from the last one in an arbitrary branch, and traversing the graph backward, and adding other branches, until the deadline violation is prevented. Finally, downsampler and upsampler nodes are inserted.

b) Permanent overload: In case of permanent overload, we can reuse the degraded version previously used, instead of computing it again.

VI. Experiments

The degradation algorithm has been implemented on a custom audio graph application written in Rust. We use libsamplerate[1] to handle the resampling. It provides 5 converters with various qualities.

In our experiments, with 2000 modulator nodes on a simple chain on a Macbook Pro with a 2.6 GHz Intel Core i7 processor and 8 Gb of RAM, the scheduler has enough time to detect that a deadline violation could occur and to degrade the graph. The overhead was on average 1,25 % of the load for 2000 nodes. However, the complexity of choosing the nodes and updating the remaining times is linear in the number of the nodes.

VII. Conclusion and perspectives

We have devised an online algorithm to degrade nodes in a dataflow graph, to reach a tradeoff between quality and lateness. In case of permanent overload, as soon as we execute the first node in the graph, we know that we have to degrade some nodes. It means that we could use pre-computed optimal degraded versions of the graph and switch to them directly. The online algorithm is still useful in case of transient overload, which is detected when executing an arbitrary node in the graph. We cannot store all possible degraded graphs for a given audio graph and so we are working on using a hybrid online and offline computation of the degraded versions of the graph. We want to validate the work on more comprehensive experiments with random graphs and also typical graphs for IMS. The next step would be to add the degradation algorithm to an existing IMS; we have started to implement it in an open source IMS, Puredata.

Acknowledgements

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF) Grant S11404-N23 and S11411-N23), by ENS Rennes, and by Inria.

References

[1] M. Puckette, “Using pd as a score language,” in *Proc. Int. Computer Music Conf.*, September 2002, pp. 184–187. [Online]. Available: http://www.crca.ucsd.edu/~msp

[2] G. Wang, “The chuck audio programming language.” a strongly-timed and on-the-fly environ/mentality”,” Ph.D. dissertation, Princeton University, 2009.

[3] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard, “Operational semantics of a domain specific language for real time musician–computer interaction,” *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 343–383, 2013.

[4] S. Akhshabi, A. C. Begen, and C. Dovrolis, “An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http,” in *Proceedings of the second annual ACM conference on Multimedia systems*. ACM, 2011, pp. 157–168.

[5] J. W. Liu, K.-J. Lin, W. K. Shih, A. C.-s. Yu, J.-Y. Chung, and W. Zhao, *Algorithms for scheduling imprecise computations*. Springer, 1991.

[6] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, “Randomized accuracy-aware program transformations for efficient approximate computations,” in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 441–454.

[7] K.-E. Årzén, V. Romero Segovia, S. Schorr, and G. Fohler, “Adaptive resource management made real,” in *3rd Workshop on Adaptive and Reconfigurable Embedded Systems*, 2011.

[8] A. Burns and R. Davis, “Mixed criticality systems-a review,” *Department of Computer Science, University of York, Tech. Rep*, 2013.

[9] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[1]http://www.mega-nerd.com/SRC/