Approximate audio processing in an audio graph for interactive music systems

Pierre Donat-Bouillud, Christoph Kirsch

Approximate audio processing in an audio graph for interactive music systems

Pierre Donat-Bouillud
ENS Rennes, Inria Mutant, Ircam STMS

Christoph Kirsch
Salzburg University

Abstract

Interactive music systems are highly dynamic systems that combine audio processing and control in real-time, and they often have to work on soft real-time platforms, where no stringent real-time guarantees can be upheld. We present here an overhead-aware online degradation algorithm that find a tradeoff between quality and lateness for the processing nodes of a dynamic audio graph. We show that we can scale to thousands of nodes.

I. Introduction

Interactive Music System (*IMS*) are highly dynamic programmable authorship systems, that combine audio processing, and control, in real-time, so that a musician and a computer can interact on stage. These IMS are mainly used by composers and musicians on mainstream operating systems.

Off-the-shelf mainstream operating systems are systems where a reliable estimation of the worst case execution time (WCET) of a task is difficult: these systems are deployed on processors with complex cache hierarchies, they rarely provide real-time schedulers, and many unpredictable tasks can interact in the system.

Hence, instead of assuming that we know the worst execution time, we adapt the execution time of a task on this kind of systems, by doing *approximate computing*. For that, given a multirate dataflow graph of processing nodes which has to be scheduled in real-time, we aim at choosing one or several nodes for which to degrade computations while preserving real-time constraints. In this framework, tasks are considered as blackboxes, and degradations can be *resampling*, or *substituting* a processing node by a another version. Modeling the tasks as tasks in a dataflow graph makes it possible to better describe degradations independently of what kind of processing the nodes are performing. Choosing a node is to be done online, during execution. If the dataflow graph is a static dataflow graph isolated from other tasks, it could be done before execution, not dynamically. However, in our case, processing nodes compete together as well as with various applications on the operating system, hence we consider a dynamic dataflow graph. It also means that no accurate profiling is possible before execution, but it has to be done online.

Considering time as a resource, it appears that in real-time systems, time is often the only resource that is degraded (by missing a deadline). Here, we also degrade other ones, and make explicit the tradeoff among various quality measures of the task (lateness, samplerate). Our scheduler is overhead-aware, and we aim at keeping the computations of the scheduler itself as low as possible.

Our contributions are the following ones:

- How to define quality in a dataflow graph
- How to choose which nodes to degrade in a graph
- Overhead-aware degradations
- Degradations that are suitable for audio streams

II. Background and motivations

A. Interactive music systems

Interactive Music Systems deal with audio streams and are used to perform music pieces in real-time. They combine signal processing, with filling audio buffers periodically and sending them to the soundcard, and controls, that can be aperiodic (such as GUI change) or periodic (a low frequency oscillator). Audio streams and controls are processed in an *audio graph* of processing nodes, which may be dynamic, *i.e.* some processing nodes can be added or removed during the performance. They are used at two different moments: the *composition*, when the artist programs a score, and the *performance*, when this score is executed for a concert.

Examples of IMS are Max [1] and PureData [2], of the Patcher family, that depicts the audio graph graphically. They make it very difficult to dynamically change the audio graph. More dynamic IMS are SuperCollider [3], or ChucK [4], that use dedicated textual languages. Antescofo [5] is an IMS dedicated to score-following, and has sophisticated synchronization strategies to coordinate a textual augmented score and a live performance played by a human musician, and as such, is highly dynamic.

IMS are available and used on mainstream operating systems, such as Windows, Mac OS, or Linux. These operating systems do not provide any strong real-time guarantees and temporal isolation, and for instance, starting a word processor may degrade the performance of an IMS.

To address these problems and the increasing complexity of the scores, IMS have rather chosen to try to raise up the available computing resources, by increasing the parallelism of their interactive scores and benefit from the pervasiveness of multicore processors. An example of such an attempt is the Supernova [6] scheduler for SuperCollider. However, these new schedulers do not parallelize the scores automatically, and requires explicit instructions, such as `poly~` in Max/MSP, and `ParGroup` with SuperNova.

Nevertheless, it appears that another direction to deal with these issues is to explore how some processing could be degraded. It is likely this has not been tackled yet for IMS because artists expect them to perfectly generate their musical ideas. Yet, we think that for an IMS such as Antescofo, which focuses on following a score, *i.e. temporal accuracy*, rather than on delivering the best audio quality, such degradations can be accepted.

B. Real-time constraints for audio

The soundcard requires audio samples to be written in its input buffer periodically. Typically, for a sampling rate of 44.1 kHz, and a buffer size of 64 samples, the audio period is 1.45 ms. The buffer size is usually a power of 2 and configurable, and ranges from as little as 32 samples for audio workstations to 2048 samples for some Android phones.

Hence audio processing has real-time constraints. It does not require hard real-time system, but has more stringent real-time requirements than video, where dropping a frame does not lead to a visible decrease in quality, and as such is used a lot in video streaming [7] protocols, for instance. On the contrary, dropping a few samples in an audio stream is immediately audible.

a) Underrun: The audio driver[1] and manager[2] usually uses a ringbuffer twice or four times the size of the audio soundcard buffer and audio applications fill up this ringbuffer. If audio applications miss a deadline and do not fill the the audio buffer quickly enough, it is called an *underrun* (or buffer *underflow*). Depending on the implementation, previous buffers are replayed (the so-called "machine gun" effect), or silence is played, leading to discontinuities in the audio stream, thus, cracky audio and clicks, as shown on Fig. 1. A large buffer size prevents underruns but in return entails a higher latency.

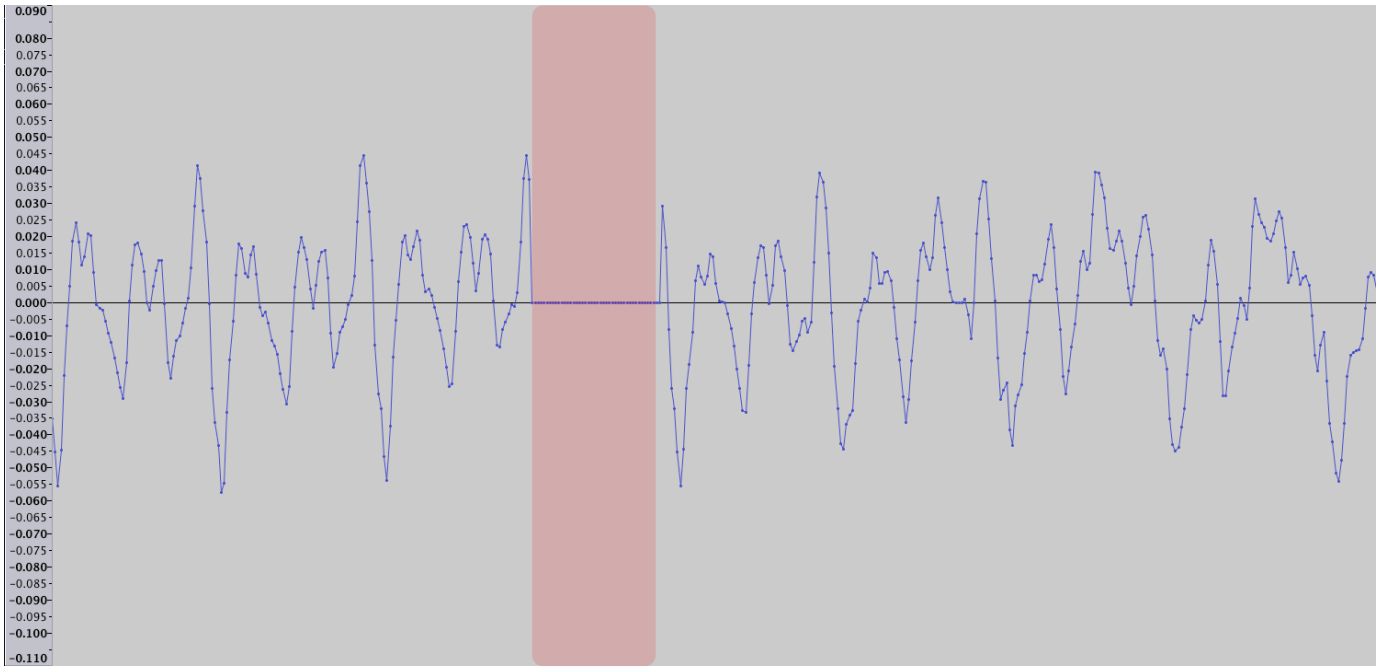


Fig. 1. The audio processing exceeds the deadline, thus cannot output any audio to the audio buffer for the soundcard. The entailed discontinuity at the read area results into a *click*.

b) Overrun: Similarly, an *overrun* occurs when the audio applications fill the audio buffer too fast for the soundcard. If the audio driver uses a ringbuffer, it will also lead to audible discontinuities in the output sound.

[1]For instance, on Linux, http://www.alsa-project.org/

[2]Jack, www.jackaudio.org ; Pulseaudio, https://www.freedesktop.org/wiki/Software/PulseAudio/ ; CoreAudio, https://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html

C. Motivations

Mainstream operating systems such as Windows or Linux are not real-time systems and do not provide guarantees on the deadlines of the audio processing computations (see Fig. 2). Audio processing has to live together with many other applications that compete for the CPU. Besides, a growing trend of interactive music systems is to port them to small boards such as Raspberry Pi, and they have to be adapted to the limited computing resources of these platforms.

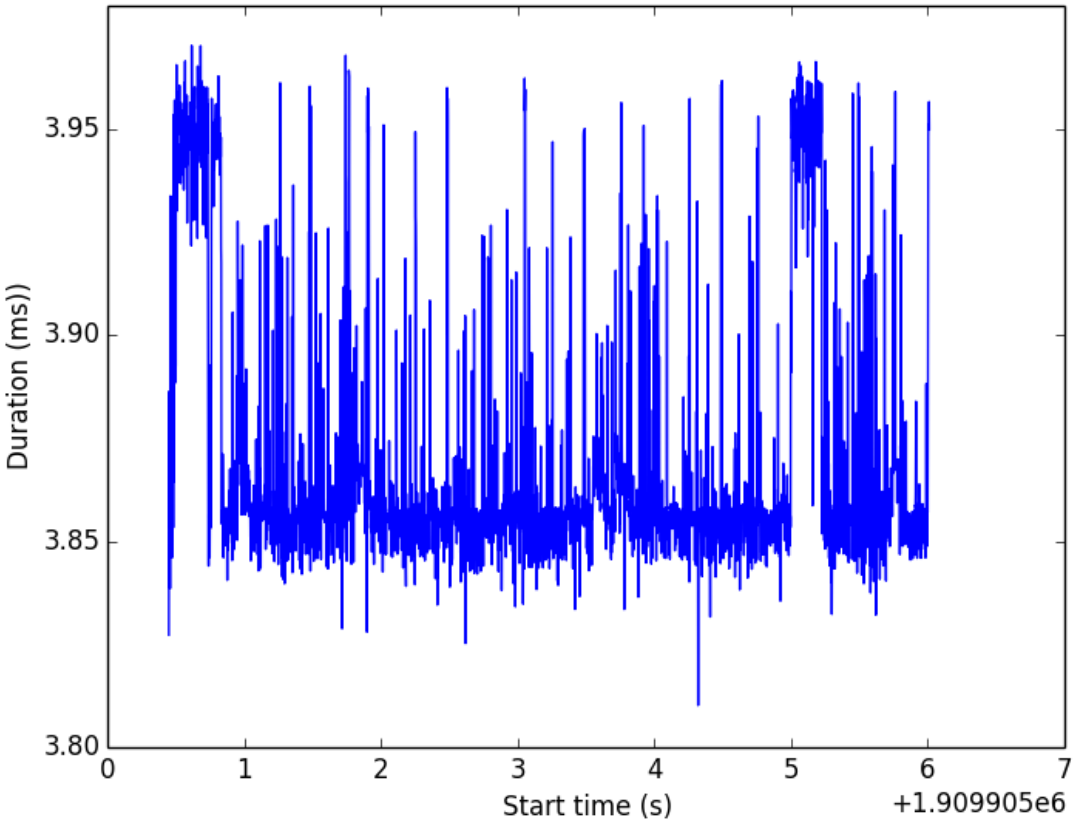


Fig. 2. Time budget given to the audio callback on Mac OS X. Although the budget is centered around 3.94 ms, there are lots of outliers, and it can vary by as far as 200 μs.

Some adaptive techniques have reached the audio community, for instance for audio streaming, but are not as widespread as techniques for video, certainly because video requires a higher throughput than audio, and because increasing latency is not problematic for audio streaming. We aim at tackling quality adaptation for interactive music system, with complex audio graph, and changes of parameters or of the graph during execution. We will prefer to resample audio instead of creating a discontinuity in the audio stream due to a deadline miss. Audio processors should be considered as blackboxes (*i.e.* programmers that provide third-party effects should not need to modify their code). That's why we will consider our work in the scope of the dataflow paradigm, where computations are described by blackbox processor nodes and tokens that flow among these nodes.

III. General model

We consider a list of tasks $T_1, \ldots, T_n$. Each task T_i has also a start time s_i, an end time t_i and a deadline d_i with $d_i > s_i$, as well as a quality measuring function $\tau(t_i) \in \mathcal{Q}$. $\mathcal{Q}$ is a totally ordered set. For $q, q' \in \mathcal{Q}$ with $q < q'$, we say that q is a worse quality than q'. The lateness l_i of T_i is $t_i - d_i$. The execution time of T_i is $t_i - s_i$.

Every task T_i is characterized by:

Maximum allowed lateness $l_i^m \geq 0$, if it is 0, it corresponds to hard real-time scheduling. if it is $+\infty$, it is soft real-time.
Worst allowed "quality" $q_i^w \in \mathcal{Q}$

We assume we have a way of measuring quality (see section V). and that we have a function τ that links lateness to quality .

The goal of the degradation algorithm is to maximize the actual quality of each tasks and minimize their lateness. In addition, the algorithm has to schedule tasks online, *i.e.* we do not know the start time of a task in advance.

In the following sections, we will add dependencies between tasks, and as a result, the quality chosen for one task can impact the quality of a dependent task. We also consider a less general task model, the dataflow model, which is well suited to represent dependencies and as a data-driven paradigm, describes well degradation on data.

IV. Dataflow graphs

Dataflow graphs are directed graphs where nodes are computations, and arcs are data paths. Data is represented as sequences of samples, called *tokens*. A node can *fire* when there is enough *tokens* in its input, and that's why the dataflow model of computation is a *data-driven* model. Nodes of the dataflow graph do not have *side effects*. In the context of audio processing, we will also call *tokens*, *samples*.

A. Synchronous dataflow

A dataflow graph is *synchronous* [8] when the number of input and output *tokens* is specified *a priori*, as shown on Fig. 3 It makes it possible to schedule the dataflow graph statically.

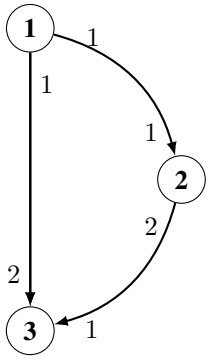


Fig. 3. A simple synchronous dataflow graph with three nodes, **1**, **2** and **3**. Data flow from **1** to **3**, from **1** to **2** and from **2** to **3**. **1** produces 1 sample per burst, and **3** needs 2 samples to be fired.

Formally, a dataflow graph is a directed graph (V, E) where V is the set of nodes, $E \subset V \times V$ the set of arcs, and a function $\mu : E \to \mathbb{N} \times \mathbb{N}$ which associates the output tokens of the producer node, and the input tokens of the consumer node. An arc (e_1, e_2) will alternatively be notated $e_1 \to e_2$.

It can also be described by an incidence matrix which includes the input and output tokens. If node j produces n tokens on arc i each time it is fired, the (i, j)-th entry in the matrix is equal to n, if node j consumes tokens, the entry is negative. If node j is not connected to arc i, then the entry is 0. This matrix is called *topology matrix*. The *topology matrix* of the graph on Fig. 3 is, if we note the arcs $\mathbf{1} \to \mathbf{2}$, $\mathbf{1} \to \mathbf{3}$, $\mathbf{2} \to \mathbf{3}$, respectively, $1, 2$ and 3 :

$$\begin{pmatrix} 1 & -1 & 0 \\ 1 & 0 & -2 \\ 0 & 2 & -1 \end{pmatrix}$$

From that, buffer sizes to keep the tokens until there are enough for one node can be computed. It also makes it possible to detect inconsistent graphs, with insufficient delays.

a) Distinguished nodes: Nodes without incoming arcs are called *input nodes*. Nodes without outgoing arcs are called *output nodes* or *sinks*. Nodes which are neither input nor output nodes are called *effects*. In Fig. 3, **1** is an input node, **3** is an output node, and **2** is an effect.

b) Execution of the graph: Let (V, E) a dataflow graph. Let T the set of tokens. A node executes on streams of tokens and produces tokens, that's to say a node e is a function $T^{\mu(a_1)} \times \cdots \times T^{\mu(a_p)} \to T^{\mu(a'_1)} \times \cdots \times T^{\mu(a'_{p'})}$ where $a_1, \ldots, a_p$ are the incoming arcs (possibly none) and $a'_a, \ldots, a'_{p'}$ the outcoming arcs of node e.

c) Dynamic dataflow [9]: Dynamic dataflow is a richer model than *synchronous dataflow*. In the *dynamic dataflow* model, the number of incoming and outcoming tokens is not fixed and can depend on the number incoming tokens. The dataflow graph itself can also change during the execution of the graph.

B. Timed dataflow

A dataflow graph per se does not describe the time instants of firing, but only their partial ordering. However, dataflow graphs are often used to describe real-time processing of data (for instance, digital signal processing).

If we assign dates of firing for input nodes and we assume a WCET for each node, we can deduce worst case time instants of firing for each nodes. If we set some worst case dates for the output nodes, we can check that these output nodes respect their deadlines.

In the following, a node e is given a worst case execution time T_e. In addition, an input node e is characterized by its start dates $(s_1^e, \ldots, s_n^e)$, $n > 0$, and an output node e, by its deadlines $(d_1^e, \ldots, d_m^e)$, $m > 0$.

We consider more specific timed nodes, which have executed periodically, with period T_e for node e. The input and output samplerates are the number of tokens consumed and produced in a period on an edge v, called respectively f_v^c and f_v^p. For an edge $v = e_1 -> e_2$ and $\mu(v) = (s_1, s_2)$, $f_v^c = \frac{s_1}{T_{e_1}}$ and $f_v^p = \frac{s_2}{T_{e_2}}$

a) Worst case execution time of a path: Given a path $e_1 \to \cdots \to e_n$, its worst case execution time is $\sum_1^n T_{e_i}$.

b) Worst case execution time of an acyclic graph: We note the paths from inputs to outputs of a graph G, $p_1, \cdots p_n$. The WCET T_G of G is $\max\{T_{p_1}, \cdots, T_{p_n}\}$.

V. Quality

The quality of an audio processing graph is a subjective and relative concept: does this version of the graph sound better or worse than this other one?

Nevertheless, there are two ways of stating more formally the quality. We can compare the output of a graph to a reference output considered as an optimum, for instance by measuring the error – this is an *a posteriori* quality measure. We can also have an *a priori* measure of quality, which would depend on some parameters of computations, such as the algorithm in use. The quality should also be a *compositional* concept: it should be possible to find out the quality of a graph given the quality of its nodes and the edges.

A. Definition

Let $\mathcal{Q}$ be a totally ordered set of qualities (finite or infinite). To each node e with p inputs and p' outputs, inputs edges $a_1, \ldots a_p$, output edges $a'_1, \ldots, a'_{p'}$, we associate a quality function $q_e : T^{\mu(a_1)} \times \cdots \times T^{\mu(a_p)} \times T^{\mu(a'_1)} \times \cdots \times T^{\mu(a'_{p'})} \to \mathcal{Q}$. Less formally, q_e compares the input data and the output data and state the quality of the output given this input.

a) Quality of a path: 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:

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

For a chain $e_1 \to \cdots \to e_n$, we will note:

$$q_{e_1 \to \cdots \to e_n} = \bigotimes_i^n q\left((e_{i-1} \circ \cdots \circ e_1(t), e_i \circ \cdots \circ e_1(t)\right)$$

with the notation $e_0 = id$. $\otimes$ is associative but not commutative in general.

We assume that $q_{e_1 \to e_2} \leq \min\{q_{e_1}, q_{e_2}\}$, that's to say, the quality never increases on a path.

b) Quality of a graph: 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. For instance, for the graph of Fig. 3, $q_{\mathcal{G}} = \min\{q_{e_1 \to e_3}, q_{e_1 \to e_2 \to e_3}\}$.

B. Degrading quality

Here, we present two ways of degrading quality. This is similar to what is presented in [10].

a) Alternate versions for nodes: The digital processing performed by the node can be substituted by other processing with a lower quality and lower worst execution time. This node e will be represented by a finite set of transformation, quality, and WCET, $\{(f_i, q_i, T_i)\}_i^e$, with the additional constraint that if $q_i < q_j$, then $T_i < T_j$.

b) Resampling: 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 rates with which samples arrive is called *resampling*. If the rate decreases, it is *downsampling* and if it increases, *oversampling*.

To represent resampling, we insert nodes in the the graph that will change the rate of producing or consuming samples, as shown on Fig. 4. These resampling nodes are normal processing nodes, and so have a quality measure and a WCET, which makes it possible to take the overhead of this degradation into account. They can also have alternative versions. For instance, a downsampler can simply output one sample very two samples. However, some frequency artifacts can occur, and more elaborated will use a low-pass filter to get rid of them in addition.

If we insert a downsampling node, all the nodes that are on a path starting on this nodes will process on a downsampled stream. Depending on the quality requirements, or anyway in the probable case of an output node dictating a specific samplerate, we also have to insert an oversampling node.

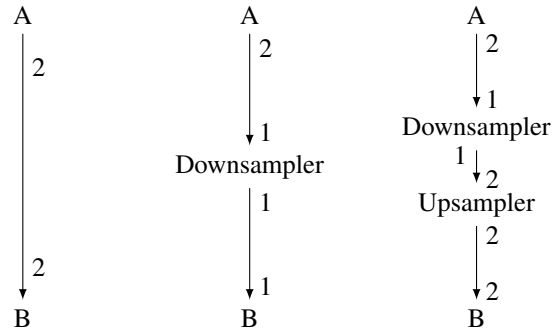


Fig. 4. On the second path, a downsampler is inserted between A and B, hence B is degraded. On the right, downsampler and an upsampler nodes are inserted between nodes A and B, maybe because B imposes a specific samplerate.

C. Measuring the quality

We precise here how we can describe q_e for a given node in practice.

a) A priori: We know that given parameters of the effect lead to the same quality.

b) A posteriori: A straightforward way to measure the quality is to compare the input signal with an output signal given by the node composed into its inverse. We can compare the distance between the two signals. As we must do that in real time, we are looking for an *instantaneous* error. We could measure it per sample, however, to get a smoother error, we use sliding windows. However, the error measurement adds up a non negligible overhead, so we have preferred to focus on *a priori* error measurement.

c) Quality when audio is output too late: In this case, the soundcard sees zeros in its input buffer after a non-zero content, which entails a discontinuity, thus a click. 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 samplerate yields a better quality than a discontinuity.

D. Overhead of resampling

We degrade a chain $\mathcal{C}$ of processing nodes $e_1, \ldots, e_n$ by *downsampling*. Let e_{down} and e_{up} the nodes that respectively downsample at the beginning of the chain, and upsample at the end of it.

The processing time of the whole chain is at least divided by the downsampling factor, as the processing time of a node is no more than linear in the number of input samples.[3] An upper bound on the whole processing time becomes:

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

VI. Degrading the whole graph

The tasks are dependent tasks, and the dependencies are given by the audio graph. We can find the schedule by performing a topological sort on the graph (which means that we assume that the audiograph is acyclic).

A. Offline algorithm

We aim at finding the best tradeoff between quality and lateness. We can state this problem as an optimization problem.

a) Optimization problem: Let $\mathcal{G}$ an audio graph with nodes $e_1, \ldots, e_n$ and $q_{\mathcal{G}}$ its associated quality measure :
maximize $q_{\mathcal{G}}$ under the constraints $\tau(e_1) \leq l_{e_1}, \cdot, \tau(e_n) \leq l_{e_n}$

b) : Provided that the quality functions have values in a continuous set (for instance, a resampling ratio), as well as the lateness ones, we can use standard optimization algorithms to solve the problem.

Other optimization problems are interesting, such as, given the quality of the graph, minimizing the lateness, or maximizing the quality while not being late at all.

B. Online algorithm

Though finding the best tradeoff between quality and lateness is tractable, in the case of a dynamic real-time audio graph, where processing nodes can be added or modified on the fly, it is too costly to solve optimally the optimization problem online, in real-time. It also requires to know the WCET of the nodes, which are not known accurately for a mainstream OS. We present here a overhead-aware degradation algorithm to react to transient overload and permanent overload.

[3]Real-time audio programmers always enforce this maximum complexity, and so we assume it here.

a) Transient overload: The first observation is that given a chain composed of the same processing nodes, it is better in general 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. V). As such, degrading nodes the nearest to an output will entail a better overall quality.

An another heuristic is to try to minimize the number of resampling nodes we add while maximizing the nodes that are degraded, in order to minimize the degradation overhead. Hence, we aim at minimizing the number of branch in the audio graph we degrade, and so we try to explore one branch at a time. We will call this algorithm *progressive* algorithm.

Every processing cycle, as shown in algorithm 1, we check before executing every node if there is enough time to process it before the deadline. If it is not the case, we look for nodes to degrade among the nodes that have not been executed yet, as shown in algorithm 2: starting from the last node in the graph, we traverse the graph backward until we have degraded enough. If one branch is not enough, we add another branch that terminates at the last node. We can finally insert upsampler and downsampler nodes at the beginning and the end of these branches.

Algorithm 1 Executing the audio graph during one cycle, with degradations.

Require: S a schedule, G an audio graph with associated execution times, d deadline
  while schedule is not empty **do**
    node ←pop_first(schedule)
    UPDATE(expectedRemainingTime)
    if expectedRemainingTime ≥ 0 **then**
      CHOOSENODES(G, deadline, expectedRemainingTime)
    end if
    samples ←GETINCOMINGSAMPLES
    if node.firstToDegrade **then**
      DOWNSAMPLE(samples)
    end if
    outBuffers ←NODE(samples)
    if node.lastToDegrade **then**
      UPSAMPLE(outBuffers)
    end if
    update performanceCounters
    node.visited ←true
  end while

Algorithm 2 How choosing the nodes to degrade.

  function CHOOSENODES(graph, budget, expectedRemainingTime)
    expectedDegradedTime ←expectedRemainingTime
    while budget - expectedDegradedTime ≤ 0 **do**
      currentNode ←LASTNODE(graph)
      LASTNODE(graph).lastToDegrade ← true
      do
        UPDATE(expectedDegradedTime)
        parentNodes ←PARENTS(currentNode)
        currentNode ←FIRSTNOTVISITED(parentNodes)
        currentNode.visited ← true
        expectedDegradedTime ← expectedDegradedTime − EXPECTEDTIME(currentNode) + DEGRADED-TIME(currentNode)
      while ¬ currentNode.visited ∧
      currentNode.firstToDegrade ← true
    end while
  end function

We also consider a simpler, *exhaustive*, algorithm, for which all the remaining (non-executed) nodes are degraded if the expected remaining time would entail a deadline miss. This algorithm degrades more than the *progressive* algorithm, but has less overhead.

b) Permanent overload: The algorithms perform as well in case of permanent overload, as they are able to degrade the whole graph. If we can detect the permanent overload, we can reuse the degradation plan chosen in the previous cycles instead of recalculating it. A permanent overload occurs when the system is overloaded for a large amount of processing cycles.

VII. Experimental validation

A. Resampling

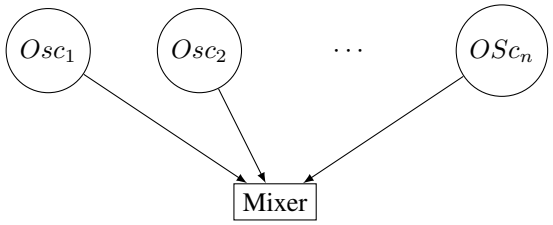
The algorithms have been implemented in rust, and with the audio graph logics, span nearly 2000 lines of code. The codes and experiments are available upon request and will be made available online after the publication of this work.

a) Resampling in practice: Secret Rabbit Code[4] (aka libsamplerate) is a high-quality opensource library to resample with arbitrary ratios, from downsampling by 256 to upsampling by 256. Another feature we use here is that the resampling ratio can be changed in real-time.

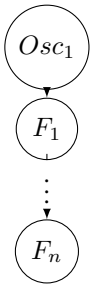
It provides 5 converters, *best, medium, fastest quality sinc converters*, based on the sinc function, as in [11]; a *zero order hold converter*, where interpolated values are equal to the last value, and a *linear converter*.

b) Quality of resampling in practice: Here, we have chosen to use a downsampling ratio of 2. We can assess the quality of resampling in two ways, *a priori*, and *a posteriori*. *A priori*, the converters previously introduced are classified from the best quality ones (and the most time-consuming) to the quickest and poor-quality ones. For every converter, the more the sampling rate, the better the quality, though the human auditory system is not able to perceive frequencies higher than 20 kHz for most people, which means that the sampling rate, per Nyquist theorem, must be at least 44,1 kHz. However, oversampling makes it possible to take into account the errors that come from audio processing, and that is why our framework takes frequencies higher than 44,1 kHz into account.

B. Results on various audiographs



(a) Flat audio graph with n oscillators.



(b) Chained audio graph with one oscillator and n effects.

Fig. 5. Graphs used for the experiment

We compare the various online algorithms on different graph shapes: a graph with depth one where all nodes, which are oscillators, are connected to the output node that mixes its input, on Fig. 5(a), and a graph with only one branch, where an oscillator is chained with modulators, on Fig. 5(b).

The experiments have been performed on a Mac Book Pro with a 2,6 GHz Intel Core i7 processor and 8Gb of RAM and we run the audio graphs for $5s$.

On Fig. 6, we show how the exhaustive online algorithm performs, for 2000 and 3000 modulators for the graph of Fig 5(b). The remaining budget is the time remaining before the deadline after all the processing has finished during one cycle. If it is negative, it means that the deadline has been missed and that the node is late. For the graph with 2000 modulators, even though the scheduler detects that it has to degrade, it is enough for the first time not to miss the deadline. *Expected time* is the expected remaining time as computed given the mean execution times of the nodes and the overhead of the resamplers, either at the beginning of the audio callback if there is no degradation, or when we detect than the expected remaining time would exceed the remaining time budget. For the 3000 modulators example, the algorithm is effective to prevent deadlines misses, thus, clicks. For the flat graph of Fig.5(a), although more resamplers are expected to be inserted as there are many more branches, there are less deadline misses, certainly because mixing all the oscillators is much less costly than modulating thousands of times the input.

[4]http://www.mega-nerd.com/SRC/index.html

Our measurements have also shown that the overhead of the scheduler for the exhaustive strategy is at most in typical cases of $50\mu s$ *i.e.* 1.25% of a deadline of $4000\mu s$ and 2000 nodes. However, the complexity of choosing updating the remaining times and updating the nodes is linear in the number of nodes in the graph.

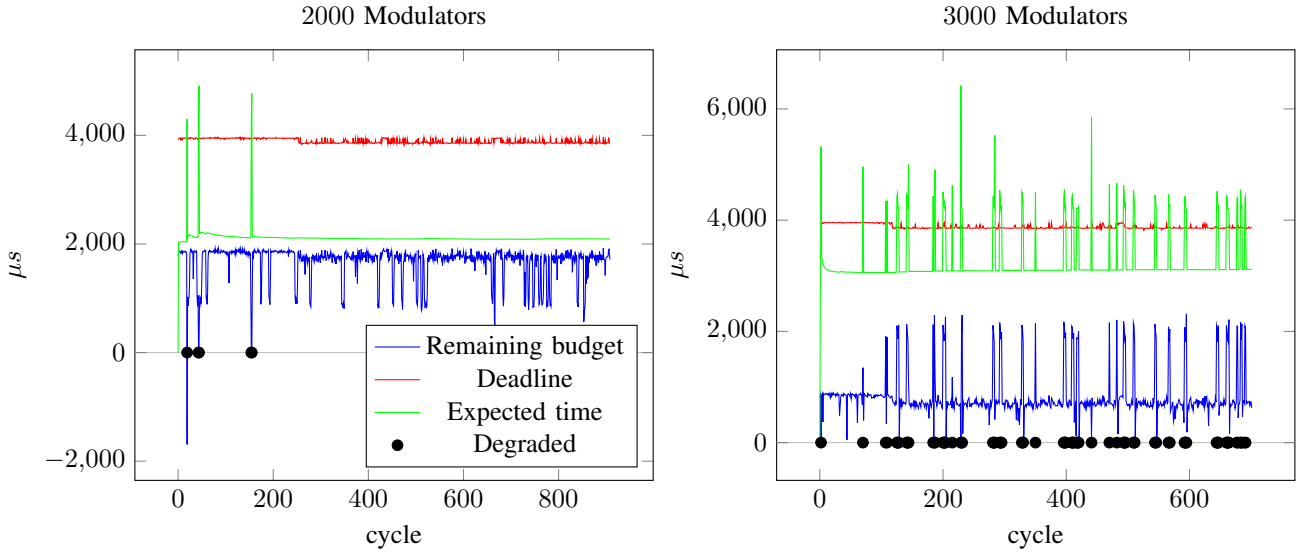


Fig. 6. Results for a chain graph such as in Fig. 5(b) for the exhaustive strategy.

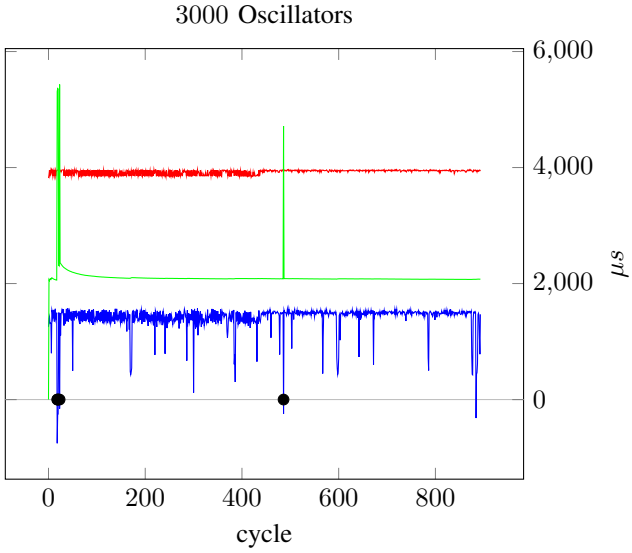


Fig. 7. Results for a flat graph such as in Fig. 5(a) for the exhaustive strategy.

VIII. Related work

Some approaches have dealt with adaptive scheduling, either by discarding tasks, or by degrading them. Some works also deal with adaptive scheduling without degrading.

A. Approximate computing

Approximate computing is a paradigm of computation that allow some errors in computations to improve performance. It relaxes the concept of correctness, to a correctness with a quantified error.

In [12], Venkataramani and al. claim that *intrinsic application resilience* stems from:

1) Not an unique answer, but a range of answers are acceptable
2) Users have got used to accept good-enough results

3) Input data is noisy, and algorithm are built to deal with this noise
4) Use of computation patterns that decrease approximations

Approximate computing can be introduced at various layers of the computing stack: circuits, architecture, softwares, but also in methodology and tools, and as a cross-layer optimization. The goal is to design systems with a favourable quality vs performance or energy tradeoff. It often needs a first profiling/training executions step and depends on an application-dependent quality measure.

a) Algorithms for scheduling imprecise computation [13]: This coarse grain strategy is to divide tasks into a mandatory part and an optional part. it makes it easier to schedule tasks in real-time. However, this model does not take into account dependencies between tasks.

b) Generating approximate computations given an error bound [10]: The computation model is based on *map-reduce*: it uses a graph to represent a program, with computation and reduction nodes. Accuracy-aware transformations are separated into two classes:

Substitution transformations They replace one implementation with another implementation. Functions have a propagation, a resource-consumption (energy, time, cost) and an accuracy specification.

Sampling transformations They randomly subsample the input of a reduction node. They are characterized by a sampling rate.

The method is to randomly choose transformations to ensure a chosen tradeoff between accuracy and resource consumption. It is used in *map-reduce* applications, and does not natively embeds time constraints[5]. It also requires a preliminary phase of profiling, and hence cannot tackle dynamic graphs.

B. Resource reservation

Here, a fraction of the CPU processing power is reserved [14] to some task. It works well for different competing tasks, such as an IMS and a word processor on the same machine. For multimedia, video tasks and audio tasks can have various reservations as audio tasks are more *hard real-time* than video tasks: a frame can be dropped in a video without perceived quality loss, whereas a click would be heard for an audio task. but does not deal with identical tasks competing for resources that would require a higher fraction of the CPU.

IX. Conclusion

We have presented an online algorithm to degrade audio computations in an audio graph. In particular, as far as we know, it is the first time that the approximate computing paradigm is used for a dataflow graph and for audio. Our degradation algorithm entails a tradeoff between quality and lateness, and scales to thousands of nodes. The quality exploration is still quite rough, and we need to have a better estimation of the audio quality, as well as taking into account a larger set of qualities.

This approximate computing scheduling is very promising for Interactive Music Systems, and we aim at experimenting it in real conditions, with highly dynamic IMSs such as Antescofo. The offline degradation strategies are also worth investigating: we could generate several audio graphs, given several combinations of latenesses, and apply them when being in a permanent overload. It should imply a better overall quality than our online algorithm but still requires some insights about the WCET of the nodes.

Acknowledgements

This work was undertaken at Salzburg university in the group of Prof. Christop Kirsch, and was funded by ENS Rennes, Inria and PHC Amadeus Let It Be.

References

[1] D. Zicarelli, "How I learned to love a program that does nothing," *Comput. Music J.*, vol. 26, no. 4, pp. 44–51, 2002.
[2] 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
[3] J. McCartney, "Supercollider: a new real time synthesis language," in *Proceedings of the International Computer Music Conference*, 1996. [Online]. Available: http://www.audiosynth.com/icmc96paper.html
[4] G. Wang, "The chuck audio programming language." a strongly-timed and on-the-fly environ/mentality"," Ph.D. dissertation, Princeton University, 2009.
[5] J. Echeveste, "Un langage de programmation pour composer l'interaction musicale," Ph.D. dissertation, Paris VI, 2015.
[6] T. Blechmann, "Supernova-a multiprocessor aware real-time audio synthesis engine for supercollider," Master's thesis, Vienna University of Technology, 2011. [Online]. Available: http://tim.klingt.org/publications/tim_blechmann_supernova.pdf
[7] 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.
[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[9] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.
[10] 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.

[5]Though it might be possible to design an aggregate error metric that also takes time into account.

[11] J. O. Smith and P. Gossett, "A flexible sampling-rate conversion method," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, vol. 9. IEEE, 1984, pp. 112–115.
[12] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Computing approximately, and efficiently," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 748–751.
[13] 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.
[14] 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.