Ordonnancement adaptatif d'un graphe audio avec dégradation de qualité

Pierre Donat-Bouillud

ORDONNANCEMENT ADAPTATIF D'UN GRAPHE AUDIO AVEC DÉGRADATION DE QUALITÉ

Pierre Donat-Bouillud
Sorbonne Université/STMS/Inria

RÉSUMÉ

Les systèmes interactifs musicaux sont des systèmes particulièrement dynamiques qui combinent du traitement du signal avec du contrôle en temps réel. Ils sont souvent utilisés sur des plateformes où il n'est pas possible d'avoir des garanties temps réel précises. Nous présentons ici un algorithme de dégradation temps réel en ligne pour obtenir un compromis entre la qualité audio et les retards par rapport aux échéances audio, dans un graphe audio dynamique. Nous évaluons expérimentalement les performances de l'algorithme.

1. INTRODUCTION

Les systèmes interactifs musicaux (SIM) [15] sont des systèmes programmables de création artistique qui combinent des traitements audio avec du contrôle en temps réel dans un graphe audio. Les graphes audio connectent des nœuds de traitement du signal dans un graphe reconfigurable en ligne et se plient à des contraintes temps réel non critiques, par rapport aux systèmes embarqués dans un avion par exemple, mais plus exigeantes que pour de la vidéo.

Compositeurs et musiciens utilisent généralement ces SIM sur des systèmes d'exploitation grand public comme Windows, macOS, ou Linux, où une estimation fiable du temps d'exécution dans le pire des cas (WCET) est difficile, à cause d'une hiérarchie complexe des caches du processeur, de l'absence d'ordonnanceurs temps-réel, du manque d'isolation temporelle entre tâches, et de la difficulté de prédire quelles tâches seront exécutées à un instant donné.

Par conséquent, nous ne pouvons pas supposer que l'on connaît le WCET des tâches mais nous pouvons réagir aux changements dans l'environnement d'exécution en modifiant le temps d'exécution d'une tâche avec le paradigme de la *programmation approximée* [17], par dégradation du graphe audio. Il ne s'agit pas de seulement dégrader un seul nœud d'un graphe audio, mais de *choisir les nœuds à dégrader*, tout en préservant les contraintes temps réel. Dans ce contexte de traitement du signal, les nœuds sont vus comme des *boites noires* et les dégradations s'effectuent seulement sur les flux de données entre les boites, en *rééchantillonnant* ces flots de données. On pourra aussi envisager de *substituer* un nœud par une autre version du nœud moins demandeuse en ressource de calcul.

Si on considère que le temps est une ressource, dans les systèmes temps-réel, le temps est souvent la seule ressource à être dégradée [7] (en ratant une échéance). Ici, nous dégradons aussi d'autres ressources et nous cherchons à trouver un compromis explicite entre diverses mesures de qualité de la tâche, comme le taux d'échantillonnage du signal.

Dans un premier temps, nous avons étudié comment choisir les nœuds à dégrader au moment de l'exécution, *en ligne.* L'ordonnanceur doit tenir compte de son propre temps de calcul et nous faisons en sorte que les calculs de l'ordonnanceur eux-même soient les plus légers possibles. Un modèle particulièrement bien adapté à décrire des tâches de traitement du signal et les flots entre les nœuds est le modèle *flot de données* [8]. Nous montrerons comment nous pouvons adapter le paradigme de *la programmation approximée* au modèle *flot de données*.

Nos contributions sont les suivantes :

— intégrer les dégradations au modèle *flot de données*
— détermination des nœuds à dégrader (en ligne)
— application de ces dégradations au cas particulier d'un graphe audio

2. CONTEXTE ET MOTIVATIONS

2.1. Les systèmes interactifs musicaux

Les SIM servent à jouer des pièces de musique, que l'on décrit par des partitions destinées à l'ordinateur, sous la forme de programme. Pour cela, ils manipulent des flux audio, au moment du concert, en temps réel, à l'aide de divers effets audio. Ils effectuent donc un traitement du signal, ils remplissent périodiquement des tampons audio, les envoient à la carte son, et permettent de contrôler les traitements, avec des contrôles apériodiques (comme des changements de l'interface graphique) ou bien périodiques (par exemple via un oscillateur basse fréquence). Les flux audio et les contrôles sont traités dans un graphe audio de nœuds de calcul audio, qui peut être dynamique, c'est-à-dire que des nœuds peuvent être ajoutés ou retirés pendant l'exécution.

Puredata [13] et Max/MSP [20] sont des exemples de SIM, qui affichent graphiquement le graphe audio mais rendent compliquée sa modification de façon dynamique comme résultat d'un calcul. D'autres SIM, comme ChucK [18] ou SuperCollider [11], sont plus dynamiques. Dans Antescofo [6], des musiciens humains et un ordinateur

Pierre Donat-Bouillud, *Ordonnancement adaptatif d'un graphe audio avec dégradation de qualité*, Journées d'Informatique Musicale (JIM 2018), Amiens, France, éd. L. Bigo, M. Giraud, R. Groult, F. Levé, pp. 25–33, 2018.

peuvent interagir sur scène en concert, à l'aide de stratégies de synchronisation sophistiquées spécifiées par une partition augmentée qui peut aussi décrire des graphes audio dynamiques [5].

2.2. Les contraintes temps-réel pour l'audio

La carte son exige que des échantillons audio soient écrits dans son tampon d'entrée périodiquement. Pour le taux d'échantillonnage d'un CD de 44.1 kHz par exemple, et une taille de tampon de 64 échantillons, la période audio est de 1.45 ms. Suivant la latence visée et les ressources de la plateforme considérée, la taille du tampon peut aller de 32 échantillons pour des ordinateurs dédiés au traitement du son, à 2048 échantillons pour certains téléphones tournant sous Android.

Les contraintes temps-réel pour l'audio sont non critiques, mais les exigences sont plus strictes que pour la vidéo. Pour la vidéo, perdre une image parmi les 24 images par seconde n'entraîne pas une baisse visible de la qualité ; de nombreux protocoles de *streaming* [1] se le permettent donc. Au contraire, rater une échéance pour une tâche audio est immédiatement audible.

Sous-alimentation du tampon Le pilote audio utilise un tampon circulaire dont la taille est un multiple de la taille du tampon de la carte son. Si la tâche audio rate une échéance, elle ne remplit pas le tampon assez rapidement. Suivant l'implémentation, les tampons précédents sont rejoués (l'effet *mitraillette*) ou du silence est joué, ce qui conduit à de l'audio avec des craquements ou des clics dus aux discontinuités dans le signal, comme on peut le voir sur la Figure 1.

Un tampon de grande taille aide à diminuer ce non-remplissage, mais augmente la latence audio.

Débordement du tampon De la même manière, dans certaines implémentations, remplir le tampon audio trop rapidement peut amener à des discontinuités audibles dans le son en sortie si les échantillons de trop ne peuvent pas être stockés pour être utilisés par la suite.

2.3. Motivations

Les systèmes d'exploitation grand public comme Windows, macOS ou Linux ne sont pas des systèmes temps-réel et ne donnent aucune garantie forte sur les échéances des traitements audio (voir la Figure 2). Les applications effectuant des traitements audio cohabitent avec de nombreuses autres applications et entrent en compétition pour une part de temps CPU. Par ailleurs, les SIM sont de plus en plus portés vers des cartes embarqués comme le Raspberry Pi et doivent s'adapter aux ressources de calcul limitées de ces plateformes.

Pour relever ces défis, et tenir compte de la complexité croissante des œuvres, les SIMs ont plutôt choisi d'exiger plus de ressources de calcul disponibles et d'augmenter le parallélisme des partitions interactives pour profiter de

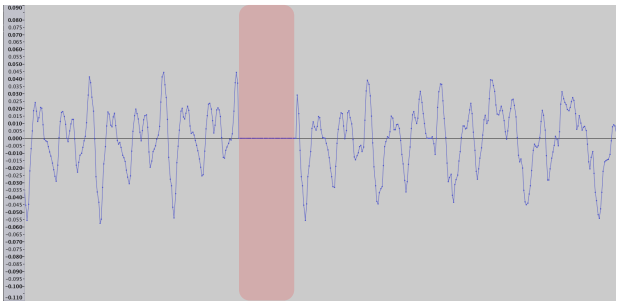


Figure 1 : Le traitement audio a raté l'échéance. Ainsi, aucun échantillon audio produit par ce traitement audio pour ce cycle audio ne peut être envoyé au tampon audio de la carte son. Dans cette implémentation, le système envoie du silence, c'est-à-dire des échantillons à zéro. Cela entraîne une discontinuité du signal à l'endroit de la bande rouge, et donc un *clic*.

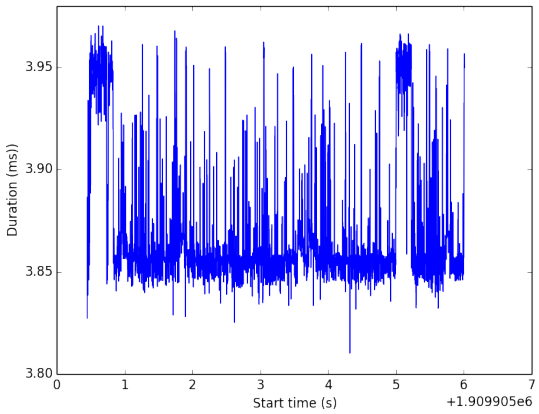


Figure 2 : La ressource temporelle accordée au *callback* audio sur un Mac Book Pro avec Mac OS X. Bien qu'elle soit centrée autour de 3.94 ms, il y a des valeurs éloignées et elle peut varier jusqu'à 200 μs.

la généralisation des processeurs multicœurs. Un exemple de système qui s'appuie sur des architectures parallèles est l'ordonnanceur de tâches Supernova [3] pour SuperCollider. Cependant, ces nouveaux ordonnanceurs ne parallélisent pas les partitions automatiquement, requièrent des instructions explicites, comme `ParGroup` pour SuperNova et `poly` pour Max/MSP, et sont donc difficiles à programmer.

Cependant, une autre façon de faire face à ces défis est d'explorer comment les traitements audio peuvent être dégradés sans que les dégradations soient perçues (ou en quantifiant ces dégradations), au moment où les ressources de calcul demandées par les pièces augmentent (chaînes audio à 96 kHz, comme dans la pièce Re Orso de Marco Stroppa [1]), en même temps que les supports – cartes embarquées, systèmes enfouis, plateformes mobiles – sur lesquels peuvent être jouées les pièces se démocratisent.

1 . http ://brahms.ircam.fr/works/work/27678/

2.4. Travaux en rapport

Certaines approches en ordonnancement ont déjà porté sur l'ordonnancement adaptatif, en temps réel critique ou non-critique, soit en supprimant complètement des tâches, soit en les dégradant, avec le paradigme de programmation approximative ou la criticité mixte. Une autre possibilité pour s'adapter aux changements des ressources de calcul disponibles pendant une exécution est d'effectuer des réservations d'une certaine capacité de calcul.

2.4.1. Le paradigme de programmation approximative

Il s'agit d'un paradigme de programmation dans lequel on autorise des erreurs dans les calculs en échange d'une amélioration des performances temporelles. La notion de correction d'un programme est relâchée pour celle d'une correction avec une erreur quantitative. Ce paradigme permet de mettre au point des systèmes dans lesquels on recherche un compromis entre une bonne qualité et les performances ou la consommation en énergie. Venkataramani et al., dans [17], proposent les critères suivants pour décider de la pertinence d'utiliser ce paradigme :

- il n'y a pas une réponse unique, mais un intervalle de réponses est acceptable ;
- les utilisateurs se sont habitués à obtenir des résultats *acceptables*, suffisamment bons mais pas parfaits ;
- les données d'entrée sont bruitées et les algorithmes qui les traitent sont conçus de façon à prendre en compte ce bruit ;
- des schémas de programmation qui diminuent les approximations sont utilisés.

Une stratégie assez basique [10] consiste à diviser les tâches du système entre une partie obligatoire et une partie optionnelle, qui peut ne pas être exécutée en cas de surcharge du processeur. Cette stratégie rend l'ordonnancement temps réel assez simple à mettre en place, avec peu de calculs supplémentaires induits par l'algorithme d'ordonnancement mais ne prend pas en compte les dépendances entre tâches, en particulier dans l'estimation de la qualité.

Un autre modèle de calcul [19] utilise un graphe pour représenter un programme de type *map-reduce*, avec des nœuds de calcul et des nœuds d'agrégation. Il génère hors ligne des versions approximatives étant donnée une certaine marge d'erreur en paramètre. Les transformations qui permettent de dégrader sont de deux types : des transformations de substitution et des transformations par échantillonnage, où l'entrée est sous-échantillonnée aléatoirement. Cependant, ce modèle est destiné à des applications de traitement par lots pour des données massives et ne prend pas en compte les contraintes temps-réel ; il requiert par ailleurs une phase préliminaire de profilage, ce qui empêche de l'appliquer à des graphes reconfigurables et dynamiques.

2.4.2. Criticité mixte

Dans les systèmes temps réel dur à criticité mixte [4], les tâches à haute criticité doivent impérativement être ordonnancées, alors que les tâches moins critiques peuvent être supprimées au cas où cela conduirait à outrepasser des échéances. Dans notre cas cependant, toutes les tâches, liées entre elles dans un graphe, ont la même criticité.

2.4.3. La réservation de ressources

Dans cette approche, une partie des capacités de calcul du processeur est réservée [2] à certaines tâches. Cela est justifié lorsque les tâches en présence sont des tâches différentes en compétition pour les ressources : dans le cas du multimédia par exemple, les tâches vidéo et les tâches audio peuvent se voir donner des réservations différentes, comme les tâches audio sont *plus temps-réel* (plus critiques) que les tâches vidéos. Cependant, la réservation de ressources n'est pas adaptée au cas où les tâches sont identiques (toutes des effets audio) et liées par des relations de dépendance de données.

3. MODÈLE DE GRAPHE AUDIO

Dans un graphe audio, des flux audio circulent des entrées aux sorties à des taux différents selon les contraintes des nœuds du graphe. Le modèle *flot de donnée* [8] est particulièrement adapté pour représenter de telles tâches avec des dépendances. Nous nous limiterons ici au modèle *flot de données synchrone*, dans lequel les taux de production et de consommation des données par les tâches sont fixes. Le modèle *flot de données* ne tient pas compte du temps. Nous rajoutons donc aux tâches des informations sur leurs dates de début et leurs durées d'exécution, ce qui constitue le modèle *flot de données temporisé*.

3.1. Le modèle *flot de données*

Le modèle *flot de données* [8] est *orienté données*. Un graphe *flots de données* est un graphe dirigé $G = (V, E)$ où les nœuds, dans l'ensemble V, représentent des calculs, en l'occurrence, des *effets audio*. Les arcs du graphe, dans $E \subset V \times V$, représentent le chemin des données et connectent les nœuds ensemble via des *ports d'entrée et de sortie*. Si v_1 et v_2 sont des nœuds, on pourra noter une arête $v_1 \rightarrow v_2$. Les données sont des séquences de *jetons* et un nœud s'exécute quand il y a suffisamment de *jetons* dans ses entrées, comme montré sur la Figure 3. Formellement, une fonction $\mu : E \rightarrow \mathbb{N} \times \mathbb{N}$ est ajoutée au graphe G, qui indique le nombre de jetons en entrée, et en sortie, d'une arête donnée. Dans le cas des traitements audio, le signal est échantillonné sur des intervalles périodiques. Les *jetons* du modèle *flot de données* correspondent à ces *échantillons* audio.

Le graphe *flot de données* est dit *synchrone* quand le nombre de jetons nécessaires à l'activation d'un nœud et le nombre de jetons produits en sortie, sont connus à l'avance.

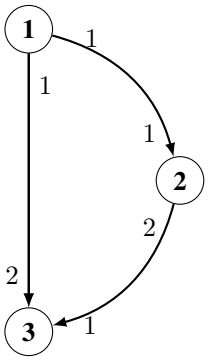


Figure 3 : Un graphe simple *flot de données synchrone* avec trois nœuds, **1**, **2** and **3**. Les données circulent de **1** à **3**, de **1** à **2** et de **2** à **3**. **1** produit 1 échantillon par exécution, et **3** requiert 2 échantillons pour s'activer. **1** est un nœud *source*, **2** un *effet*, et **3** une *sortie*.

Les nœuds sans port d'entrée sont appelés *sources* ou *entrées* : ce sont les sources sonores, comme l'entrée micro, un fichier son, un synthétiseur. Les nœuds sans port de sortie sont appelés *sorties* ou *puits*, par exemple, la sortie vers la carte son et des haut-parleurs. Les nœuds qui ne sont ni des *entrées* ni des *sorties* sont appelés *effets*.

Le modèle *flot de données dynamique* [9] est un modèle plus riche que le modèle *synchrone*, où le nombre de jetons en entrée et en sortie n'est pas fixé et peut dépendre du nombre de jetons en entrée. Le graphe peut lui-même changer au cours de son exécution.

3.2. Le modèle *flot de donnée temporisé*

Un graphe *flot de données* ne décrit pas les instants où les nœuds du graphe s'activent mais seulement leur ordre partiel. Cependant, les graphes *flots de données* représentent souvent des traitements temps réel, par exemple les traitements audio temps réel. Pour un CD, dont le taux d'échantillonnage des morceaux est de 44.1 kHz, il y a un échantillon toutes les 22,6 μs. Si on donne des dates d'activation aux *sources*, que l'on connaît un temps d'exécution dans le pire des cas ou en moyenne de tous les nœuds, on peut en déduire des temps de déclenchement de chacun des nœuds du graphe. Si par ailleurs les *sorties* ont des échéances, cela permet de vérifier que les échéances sont respectées dans le pire des cas ou en moyenne.

On ajoute donc des dates de déclenchement d'exécution pour chaque *source* v, $s_1^v, \ldots, s_n^v$ avec $s_i^v < s_{i+1}^v$ et tous les nœuds du graphe reçoivent un temps d'exécution dans le cas moyen M_v, dans le pire cas, W_v et un temps d'exécution actuel T_v. On adjoint aussi à chaque *sortie* v des dates *d'échéances* $d_1^v, \ldots, d_n^v$, avec $d_i^v < d_{i+1}^v$, qui correspondent aux moments où la carte son devra avoir reçu un certain nombre d'échantillons. Les traitements audio sont périodiques donc on suppose que pour une source v, $s_2^v - s_1^v = s_{i+1}^v - s_i^v$, et pour une sortie v, $d_2^v - d_1^v = d_{i+1}^v - d_i^v$. Les taux d'échantillonnage en entrée f_e^c et en sortie f_e^p sur une arête $e = v_1 \to v_2$ et $\mu(e) = (j_1, j_2)$, qui correspondent au nombre de jetons produits et consommés sur une période sur l'arc, est $f_e^c = \frac{j_1}{T_{e_1}}$ and $f_e^p = \frac{j_2}{T_{e_2}}$.

Le temps d'exécution dans le pire cas (respectivement en temps moyen) est, pour un chemin $v_1 \to ... \to v_n$ du graphe, $\sum_1^n W_{v_i}$ (resp. $\sum_1^n M_{v_i}$).

4. QUALITÉ

La qualité d'un graphe audio est un concept relatif et subjectif : cette version du graphe sonne-t-elle mieux ou moins bien que telle autre ? Une première façon est de calculer la différence entre une version dégradée et une version considérée comme optimale, une mesure de qualité *a posteriori*. Ce n'est pas souhaitable lors d'une exécution temps réel ; cela reviendrait à calculer simultanément deux versions du graphe, alors que le but est d'alléger les calculs. Nous calculons plutôt une mesure de qualité *a priori* qui se base sur des paramètres des calculs en cours.

4.1. Quelques propriétés d'une mesure de qualité

La mesure de qualité doit aussi être une mesure *compositionnelle*, au sens où la qualité du graphe doit être calculable comme une fonction de la qualité de ses nœuds et de ses arêtes. Chaque nœud v se voit attribuer une mesure de qualité $q_v \in [0, 1]$, 0 correspondant à la pire qualité, 1 à la meilleure.

La qualité $q_{v \to v'}$ sur un arc $v \to v'$ suit les propriétés suivantes :

Associativité $q_{v \to v'} = q_v \otimes q_{v'}$ où $\otimes$ est un opérateur associatif (et pas commutatif en général).

Décroissance $q_{v \to v'} \leq \min\{q_v, q_{v'}\}$, c'est-à-dire que la qualité n'augmente jamais sur un chemin.

De même, pour un graphe $\mathcal{G}$, la qualité $q_\mathcal{G}$ est :

$$q_\mathcal{G} = \bigotimes_{c \in \mathcal{C}} q_c$$

où $\mathcal{C}$ est l'ensemble des chemins de $\mathcal{G}$.

Elle n'est jamais supérieure à la qualité de chacune des chaînes du graphe. Par exemple, pour le graphe de la Figure 3 :

$$q_\mathcal{G} \leq \min\{q_{v_1 \to v_3}, q_{v_1 \to v_2 \to v_3}\}$$

4.2. Dégrader la qualité en pratique

On considère ici deux façons de dégrader la qualité du graphe : en insérant des nœuds pour sous-échantillonner des chemins du graphe, ou en modifiant des nœuds du graphe et en les remplaçant par une version de moindre qualité.

Insertion de nœuds de rééchantillonnage. Dans un graphe *flot de données*, les nœuds reçoivent des échantillons puis les traitent quand ils en ont reçu suffisamment. Par conséquent, si un nœud reçoit des échantillons moins souvent, il utilisera moins de temps de calcul. Changer le nombre

d'échantillons par unité de temps est une opération habituelle en traitement du signal, et est appelé *rééchantillonnage* : obtenir moins d'échantillons correspond à *sous-échantillonner*, tandis qu'en obtenir plus, à *sur-échantillonner*. Pour rééchantillonner, nous insérons des nœuds dans le graphe qui vont changer les taux de production ou de consommation des échantillons. Ces nœuds insérés sont des nœuds d'effet comme les autres ; ils peuvent interpoler des valeurs, copier des valeurs entre des tampons audio. Ils sont donc aussi munis d'une mesure de qualité et de caractéristiques temporelles, qui permettent de prendre en compte l'impact des calculs menant à la dégradation elle-même dans la décision de dégrader ou pas.

Si l'on insère un nœud sous-échantillonneur, tous les nœuds qui sont sur un chemin qui commence sur ce nœud vont opérer sur des flux sous-échantillonnés. Si le chemin se termine par une *sortie* qui dicte un taux d'échantillonnage particulier, il faut aussi insérer un nœud sur-échantillonneur.

Remplacement d'un nœud. Le traitement audio opéré par un nœud peut être remplacé par un autre traitement de qualité inférieure et de temps de calcul dans les cas pire et moyen inférieurs, c'est-à-dire, si on note v' et v'' deux versions du nœud v, si $q_{v'} < q_{v''}$ alors $M_{v'} < M_{v''}$ et $W_{v'} < W_{v''}$. Par exemple, les nœuds de sous-échantillonnage peuvent utiliser un algorithme de plus ou moins bonne qualité : garder un échantillon sur deux ou bien, en plus de cela, utiliser un filtre passe-bas pour se débarrasser des artefacts fréquentiels dus à la première méthode. Un nœud de réverbération peut être implémenté en haute qualité avec une coût de calcul important à l'aide d'une convolution avec une réponse impulsionnelle d'une salle, ou avec des lignes de retards, ce qui entraîne une moins bonne qualité (une réverbération moins *réaliste*) mais un temps de calcul rapide.

4.3. Mesurer la qualité

On choisit comme mesure de qualité le taux d'échantillonnage : plus le taux d'échantillonnage est bas, plus la qualité est basse. Quand l'audio est envoyé trop tard au tampon de sortie, on entend un clic (due une discontinuité du signal, voir Section 2). Au contraire, si le flux audio a été sous-échantillonné, le tampon de sortie comportera des échantillons non-nuls en général. Nous partons donc du principe que sous-échantillonner permet d'obtenir une meilleure qualité que rater une échéance audio.

La qualité q_v d'un nœud inséré d'échantillonnage v est $q_v = \arctan(\alpha f)$ où f est la fréquence d'échantillonnage. Le choix de la fonction $\arctan$ [2] et du coefficient α est permet de prendre en compte que d'un point de vue psychoacoustique [14], au delà d'une certain fréquence, aucune amélioration de qualité n'est perceptible par des humains. En effet, le système auditif de la plupart des êtres humains ne peut pas percevoir des fréquences au-delà de 20 kHz. Le théorème de Shannon indique donc que le taux d'échantillonnage doit être d'au moins le double, donc à peu près la fréquence des CD de 44.1 kHz. Cependant, le suréchantillonnage permet de mieux prendre en compte les erreurs de calcul et d'approximations lors des traitements audio et c'est pourquoi nous tenons quand même compte des fréquences supérieures à 44.1 kHz. On pourra prendre $\alpha = \frac{\tan(0.9)}{44.1 \cdot 10^3}$. On considère par par ailleurs que les autres nœuds sont de qualité optimale, c'est-à-dire, $q = 1$.

On considère une chaîne $\mathcal{C}$ de nœuds $v_1 \to \cdots \to v_n$ à dégrader. On note v_{sous} et v_{sur} respectivement le nœud de sous-échantillonnage et celui de sur-échantillonnage. Après insertion de ces nœuds, la chaîne devient : $v_{\text{sous}} \to v_1 \to \cdots \to v_n \to v_{\text{sur}}$. On peut prendre $\otimes$ le produit sur les réels, ou $\otimes = \min$. Comme une qualité q est toujours dans $[0, 1]$, avec $\times$, la qualité ne peut qu'effectivement diminuer sur une chaîne.Pour $\min$, intuitivement, cela indique que la qualité du tout est celle du maillon le plus faible et que l'on peut dégrader beaucoup plus de nœuds pour respecter les échéances tout en conservant la même qualité.

Les traitements audio nécessitent en général de parcourir au moins une fois chacun des échantillons du tampon audio donc ont au moins une complexité linéaire. Le temps d'exécution de la chaîne est donc au moins divisé par le facteur de sous-échantillonnage r (par exemple si l'on passe de 96 kHz à 48 kHz, $r = 2$). Le temps actuel de calcul (resp. moyen, dans le pire des cas) est majoré par :

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

5. CHOISIR LES NŒUDS À DÉGRADER

Le choix des nœuds à dégrader peut être considéré comme un problème d'optimisation. Ici, on s'intéresse au cas moyen, comme les SIM s'exécutent sur des plateformes grand public sans garanties temps-réel fortes. Le raisonnement serait le même dans le cas le pire.

Soit $\mathcal{G}$ est un graphe audio composés des nœuds $v_1, \ldots, v_n$ et $q_{\mathcal{G}}$ sa mesure de qualité associée. L'échéance d'une tâche est notée d_v ; on peut l'obtenir grâce aux dépendances entre tâches, aux dates d'activation des sources et à leur temps d'exécution en moyenne. On suppose donné $\tau : [0, 1] \to \mathbb{R}^+$ qui décrit le temps d'exécution moyen d'une tâche en fonction de sa qualité.

Le problème d'optimisation sous contraintes peut ainsi s'écrire :

maximiser $q_{\mathcal{G}}$ sous les contraintes :

$$\tau(v_1) \leq d_{v_1}, \ldots, \tau(v_n) \leq d_{v_n}$$

En supposant que les qualités ont des valeurs dans un ensemble continue, on peut essayer des algorithmes standards d'optimisation pour résoudre le problème. On sait aussi que $\tau(n) = O(n)$ comme indiqué en section 4.3.

2 . $\arctan$ admet la droite $y = 1$ comme asymptote en $+\infty$, ce qui permet de modéliser que l'augmentation de la qualité est de moins en moins grande et ne dépasse pas un palier. D'autres fonctions croissantes avec une asymptote horizontale en $+\infty$ pourraient faire l'affaire.

Cependant, le temps de calcul pour la recherche de solutions exactes n'est pas raisonnable s'il doit être fait au moment de l'exécution du graphe audio, mais est beaucoup plus pertinent pour pré-calculer des version dégradées du graphe. Ce calcul hors-ligne de versions dégradées fera l'objet d'un travail ultérieur. Ici, nous nous attachons plutôt à utiliser des heuristiques simples et rapides à calculer lors de l'exécution du graphe audio.

Exécution du graphe audio

Les tâches sont des tâches dépendantes et les dépendances entre tâches sont données par le graphe audio. On peut trouver un ordonnancement de ces tâches en effectuant un tri topologique du graphe (ce qui signifie que l'on suppose que le graphe audio est acyclique). Avant d'exécuter chaque tâche, on estime le temps d'exécution qui reste avant la fin de l'exécution du graphe, en utilisant le temps d'exécution dans le cas moyen des nœuds. Il faut ensuite vérifier si le temps restant estimé est plus petit que le temps restant avant l'échéance de la procédure audio qui envoie le signal à la carte son. Le temps d'exécution moyen du nœud est ensuite mis à jour.

Deux situations de surcharge du processeur peuvent se produire, une surcharge du processeur *transitoire*, ou bien une surcharge *permanente*. On dit que le processeur est surchargé de manière permanente quand le processeur est surchargé pendant plus de k cycles audio. La façon dont est déterminé k est hors-sujet dans le cadre de cet article.

En cas de surcharge *transitoire*. Étant donnée une chaîne de nœuds de traitement, il est préférable de dégrader les nœuds à la fin de la chaîne plutôt qu'au début, comme la qualité n'augmente jamais sur une chaîne (voir Section 4.1).

Une autre heuristique est de minimiser le nombre de nœuds de rééchantillonnage ajoutés, tout en maximisant le nombre de nœuds dégradés, pour minimiser le temps supplémentaire de calcul dû aux nœuds ajoutés. Cela implique que le nombre de chemins à dégrader doit être minimisé, et c'est pour cela qu'on choisit d'explorer le graphe branche par branche.

A chaque cycle d'exécution du graphe audio, l'algorithme 1 vérifie le temps restant à exécuter avant l'échéance et le compare au temps restant estimé d'exécution. S'il estime qu'il ne reste pas suffisamment de temps, on doit choisir parmi les nœuds restant à exécuter ceux à dégrader, comme décrit dans l'algorithme 2. On part d'un des nœuds *sortie*, on traverse le graphe en arrière jusqu'à ce qu'on a dégradé suffisamment pour que l'estimation du temps restant passe sous le temps restant donné par l'échéance. On peut explorer d'autres branches de même si ce n'est pas suffisant. Finalement, on ajoute les nœuds de sous-échantillonnage et de sur-échantillonnage au début et à la fin de ces branches du graphe.

Cependant, explorer successivement des branches, choisir le nœud à partir duquel dégrader, peut être encore trop coûteux en temps de calcul. Au lieu de l'algorithme 2, on peut plus simplement dégrader directement tous les nœuds restants du graphe, en insérant au début de chaque branche restante et à la fin les nœuds rééchantillonneurs.

Algorithm 1 Exécution du graphe pendant un cycle, avec éventuelle dégradation.

```
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
```

En cas de surcharge *permanente*. Dans le cas d'une détection d'une surcharge permanente du processeur, une version dégradée calculée précédemment peut être utilisée, sans avoir besoin de la calculer à nouveau. Les versions dégradées calculées hors-ligne pourraient être utilisées ici.

6. ÉTUDE EXPÉRIMENTALE

Les algorithmes de dégradation en ligne ont été implémentés dans un prototype en Rust. Le rééchantillonnage est effectué avec libsamplerate [3] . Cette bibliothèque *opensource* permet de rééchantillonner avec des ratios arbitraires, compris entre un sous-échantillonnage par 256, jusqu'à un sur-échantillonnage par 256. Le taux d'échantillonnage peut être changé en temps réel. La bibliothèque fournit cinq rééchantillonneurs, *best, medium, fastest quality sinc* rééchantilloneurs (comme dans [16]), *zero order hold* où les valeurs interpolées sont égales à la dernière valeur, et un *rééchantillonneur linéaire*. Ils sont ici rangés en ordre décroissant de qualité.

On considère deux formes extrêmes de graphes audio, l'un où tous les effets sont connectés à un même nœud de sortie, sur la Figure 4a, et l'autre avec une seule branche, sur la Figure 4b. Les effets utilisés sont des oscillateurs sinusoïdaux, des modulateurs par une fréquence sinusoïdale, et des mixeurs.

Les expériences ont été effectuées sur un Mac Book Pro avec un processeur Intel Core i7 à 2.6 GHz, muni de 8

3. http://www.mega-nerd.com/SRC/

Algorithm 2 Comment choisir les nœuds à dégrader.

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

Gb de Ram. Les graphes ont exécutés pendant 5 secondes chacun.

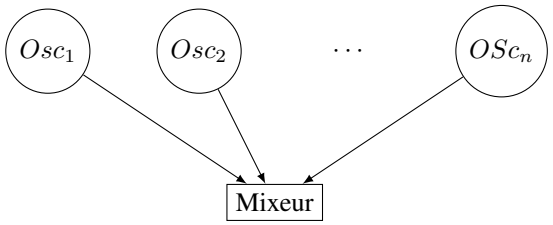
Sur la Figure 5, on montre comment l'algorithme de dégradation utilisant l'heuristique de dégradation de tous les nœuds restants se comporte quand on passe de 2000 à 3000 modulateurs, pour le graphe de la Figure 4b. La ressource temporelle restante est le temps qui reste avant l'échéance après que tous les traitements audio ont été effectués dans un cycle. Si elle est négative, cela signifie que l'échéance a été ratée et que le nœud de calcul est en retard. Pour un graphe avec 2000 modulateurs, même si l'ordonnanceur détecte qu'il doit parfois dégrader, la ressource temporelle est suffisante pour ne pas rater d'échéance en général. Pour le graphe avec 3000 modulateurs, l'algorithme de dégradation empêche bien en pratique de rater des échéances, et donc d'entendre des clics audio.

Nous avons aussi mesuré les *frais* causés par l'ordonnanceur, c'est-à-dire le temps supplémentaire pris par l'ordonnanceur pour trouver les nœuds à dégrader. Au plus, nous avons trouvé que ce temps supplémentaire s'élevait à $50\mu s$, c'est-à-dire à $1,25\%$ de l'échéance de $4000\mu s$, pour 2000 nœuds. Cependant, la complexité dans le pire des cas de l'algorithme est linéaire en le nombre de nœuds.

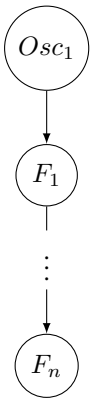
7. CONCLUSION ET PERSPECTIVES

Nous avons mis au point un algorithme en ligne de dégradation de nœuds dans un graphe flot de données dédié au son, cherchant un compromis entre qualité sonore dépendant du taux d'échantillonnage sur des branches du graphe et respect des échéances induites par la procédure d'entrée/sortie audio. En cas de surcharge permanente du processeur, une version dégradée pré-calculée du graphe pourrait être utilisée à la volée.

Nous souhaitons dans la suite de notre travail explorer



(a) Graphe audio plat avec n oscillateurs.



(b) Audio graphe chainé avec un oscillateur suivi de n modulateurs.

Figure 4 : Les graphes utilisés pour les expériences.

plus en détails le calcul de versions dégradées du graphe hors-ligne. L'algorithme en ligne utilise des heuristiques pour choisir les nœuds à dégrader, au lieu de pré-calculer exactement des versions dégradées du graphe pour différents niveaux de qualité. Les versions dégradées pré-calculées du graphe audio qui peuvent être stockées sont en nombre limité. Les surcharges transitoires du processeur peuvent arriver en pleine exécution du graphe audio, et les versions pré-calculées ne sont pas forcément adaptés car elles peuvent exiger de dégrader certains nœuds qui ont pourtant déjà été exécutées en version normale lors du cycle en cours.

Nous voulons aussi évaluer d'autres heuristiques, sur un plus grand nombre de graphes, par exemple des graphes générés aléatoirement, et tester nos algorithmes de dégradation sur de vraies pièces musicales et dans des SIMs existants ; nous avons commencé à étudier la possibilité de les ajouter dans Puredata dans le code source du SIM. Nous pourrions aussi générer directement des versions *optimisés* de patchs Puredata ou Max, ou de programmes Faust [12], en ayant inséré des sous-échantillonneurs dans le graphe audio décrit.

8. REMERCIEMENTS

Ce travail a débuté lors d'une visite dans l'équipe du professeur Christoph Kirsch de l'université de Salzbourg en Autriche, que je souhaite remercier pour ses conseils

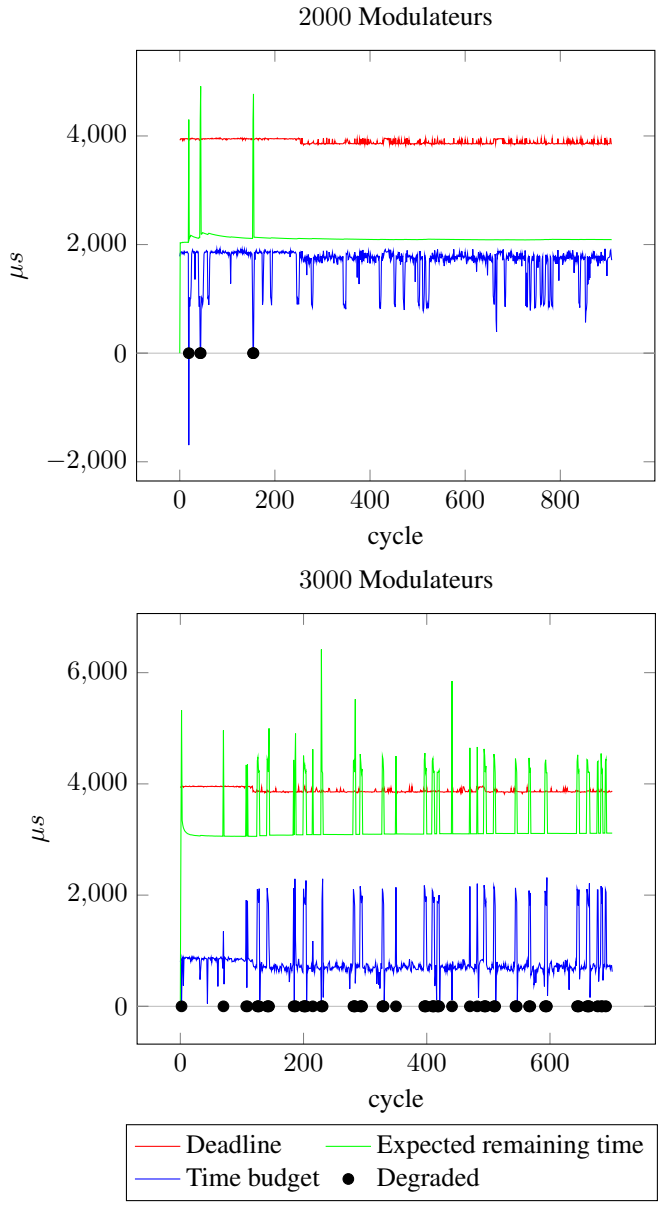


Figure 5 : Les résultats pour le graphe chainé de la Figure 4b.

avisés. Je remercie aussi mes encadrants de thèse, Florent Jacquemard et Jean-Louis Giavitto, pour leurs conseils.

9. REFERENCES

[1] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168. ACM, 2011.

[2] Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr, and Gerhard Fohler. Adaptive resource management made real. In *3rd Workshop on Adaptive and Reconfigurable Embedded Systems*, 2011.

[3] T. Blechmann. Supernova-a multiprocessor aware real-time audio synthesis engine for supercollider. Master's thesis, Vienna University of Technology, 2011.

[4] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[5] Pierre Donat-Bouillud, Jean-Louis Giavitto, Arshia Cont, Nicolas Schmidt, and Yann Orlarey. Embedding native audio-processing in a score following system with quasi sample accuracy. In *ICMC 2016-42th International Computer Music Conference*, 2016.

[6] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Operational semantics of a domain specific language for real time musician–computer interaction. *Discrete Event Dynamic Systems*, 23.

[7] Christoph M Kirsch. Principles of real-time programming. In *International Workshop on Embedded Software*, pages 61–75. Springer, 2002.

[8] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75.

[9] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83.

[10] Jane WS Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. *Algorithms for scheduling imprecise computations*. Springer, 1991.

[11] James McCartney. Supercollider : a new real time synthesis language. In *Proceedings of the International Computer Music Conference*, 1996.

[12] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9) :623–632, 2004.

[13] M. Puckette. Using pd as a score language. In *Proc. Int. Computer Music Conf.*, pages 184–187, September 2002.

[14] Stuart Rosen and Peter Howell. *Signals and systems for speech and hearing*, volume 29. Brill, 2011.

[15] Robert Rowe. *Interactive Music Systems : Machine Listening and Composing*. AAAI Press, 1993.

[16] Julius O Smith and Phil Gossett. A flexible sampling-rate conversion method. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 112–115. IEEE, 1984.

[17] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Computing approximately, and efficiently. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 748–751. EDA Consortium, 2015.

[18] G. Wang. *The ChucK audio programming language." A strongly-timed and on-the-fly environ/mentality"*. PhD thesis, Princeton University, 2009.

[19] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, volume 47, pages 441–454. ACM, 2012.

[20] David Zicarelli. How I learned to love a program that does nothing. *Comput. Music J.*, 26(4) :44–51, 2002.