Master 2 ATIAM
Mémoire de Stage

Étude du contrôle adaptatif de processus dynamiques dans les systèmes multimédia interactifs

Auteur

Samuel Bell-Bell

Sous la direction de

Dimitri Bouche
Jean-Louis Giavitto

Équipe Représentations Musicales
Institut de Recherche et de Coordination Acoustique / Musique

2 Février 2016 - 2 Août 2016

Table des matières

1 Préambule et remerciements

"Le principe de l'évolution est beaucoup plus rapide en informatique que chez le bipède." Jean Dion.

"D'une façon générale, composer un morceau consiste à écouter la musique que l'on a en soi." Bert Jansch

Je remercie en premier lieu Jean-Louis Giavitto et Dimitri Bouche sans qui cette belle exploration et cette étude n'auraient pas pu être possible. Leur encadrement et leur altruisme ont été des plus intéressants, des plus agréables et des plus efficaces tout au long de ce projet, tout en me laissant une grande indépendance.

Je remercie Jean Bresson pour ses conseils concernant ce rapport de stage et pour m'avoir prêté son bureau tout au long de mon stage alors qu'il était à l'étranger.

Je remercie Jérôme Nika, Héliante Caure et Dimitri d'avoir partagé avec moi le bureau A107 dans une ambiance conviviale et à la fois sérieuse.

Je souhaite aussi remercier Carlos Agon et Philippe Esling pour leurs conseils, pendant l'ensemble de mes études supérieures.

Je remercie les équipes pédagogique et administrative du Master 2 ATIAM pour faire de ce M2 un parcours d'exception et passionnant.

Je remercie les enseignants-chercheurs du Master STL de l'UPMC qui m'ont appris à m'adapter à de nouveaux langages rapidement lors de mon M1 et je remercie l'ensemble des enseignants de Paris VI qui m'ont permis de réaliser toutes sortes de programmes durant l'enseignement qui m'a été dispensé.

Je remercie l'IRCAM et l'ensemble des personnes y travaillant de partager leur passion pour la musique à travers la recherche et la composition, et de faire vivre cet institut dans une atmosphère accueillante.

Je remercie mes camarades de la promotion 2015/2016 ATIAM d'avoir partagé cette année pleine d'enseignements et de défis avec autant de ferveur.

Je remercie ma famille et mes amis de m'avoir soutenu tout au long de ces années.

Résumé

Les systèmes multimédias modernes, du fait de leur caractère de plus en plus interactif, rendent difficile le rendu en temps réel des données qu'ils contiennent. En effet, les interactions avec ces données peuvent les modifier en temps réel, ce qui rend l'ordonnancement statique des actions du système non adapté. Cependant, les méthodes d'ordonnancement dynamique ne permettent pas d'utiliser au mieux les ressources car elles n'utilisent aucune connaissance a priori.

L'étude effectuée lors de ce stage concerne la possibilité d'adapter en temps réel ces environnements grâce à un outil d'analyse. Les logiciels de Composition Assistée par Ordinateur (CAO) permettent désormais le calcul de processus compositionnels en temps réel. Leurs sorties prennent l'aspect de partitions musicales interactives pouvant être amenées à communiquer avec d'autres logiciels. Il faut aussi prendre en compte le fait que ces logiciels cohabitent avec des programmes en cours d'exécution dans le même ordinateur.

La recherche de stratégies d'ordonnancement pour de tels systèmes évolue. Les stratégies d'ordonnancement naïves des actions de ces systèmes sont peu efficaces. Elles doivent respecter des contraintes du temps réel lors du rendu des données qui peuvent déclencher des calculs longs venant du domaine du temps différé. Le tout devant coexister dans un même contexte.

Nous proposons un outil d'observation du comportement de tels systèmes. L'utilisation de déductions à partir de statistiques émises par cet outil ouvre la voie à de nouvelles stratégies adaptatives.

Mots-clés : CAO, OpenMusic, API de mesure et statistique, ordonnancement, multimédia, système interactif, adaptabilité, processus, multithreading, DAG.

Abstract

The modern multimedia system by their characterization more and more interactive, make it difficult to render in real-time data they contain. In fact, these interaction can modify data in real time, that make the static scheduling unadapted. However, dynamic scheduling methods don't ensure the best use of the available resource because they don't use a priori knowledge.

The study carried out at this stage is the possibility of adapting these real-time environments with an analysis tool. Computer Aided Composition software (CAC) now allow the calculation of compositional processes in real-time. Their outputs have the appearance of interactive musical scores which can be brought to communicate with other programs. We must also take into account the fact

that these software products coexists with running programs in the same computer. The research of scheduling strategies for such systems evolve. The naive scheduling strategies of actions by these systems are weak. They must respect the constraints of real-time when they rendering data that can trigger long calculations from the field of deferred time. All of this must coexist in the same context .

We offer an observation tool for the behavior of such systems. Using inferences from statistics issued by this tool opens the door to new adaptive strategies

Keywords : CAC, OpenMusic, measurement and statistical API, scheduling, multimedia, interactive systems, adaptability, thread, multithreading, DAG.

2 Introduction

Ce mémoire résume le stage réalisé dans le cadre Master 2 ATIAM 2015/2016. Ce master est une formation pluridisciplinaire aux sciences de la musique (Acoustique, Traitement du signal, Informatique, Appliqué à la Musique) est proposé par l'Université Pierre et Marie Curie. De plus ce master est organisé en collaboration avec l'IRCAM (Institut de Recherche et Coordination Acoustique/Musique) et Télécom ParisTech qui accueillent l'enseignement. L'équipe Représentations Musicales propose d'encadrer ce stage. Il est réalisé à l'occasion d'une étude relative au contrôle adaptatif de processus dynamiques dans les systèmes multimédias interactifs.

Le logiciel de composition OpenMusic (OM) appartient à la famille des logiciels de CAO. Il s'agit de l'environnement où le stage se déroule. L'étude réalisée intéresse aussi l'équipe MuTant dans le cadre du développement d'Antescofo qui est un logiciel de suivi de partition. En effet, malgré la granularité temporelle plus précise dans Antescofo que dans OM, le comportement du système face à différents travaux reste a priori le même (les exécutions en temps réel restant aléatoires).

OM permet la réalisation d'idées musicales sous forme de processus produisant des résultats en temps différé. Ces résultats sont issus d'une phase de calcul et sont considérés comme "statiques" puisqu'on ne peut pas interagir avec eux lors de leurs exécutions.

De récents travaux dans OM et dans Antescofo ouvrent la CAO au temps réel. Ces travaux ont notamment entrainé une évolution interne du langage OM grâce au développement d'un paradigme de programmation réactive. Ce paradigme vise à conserver la cohérence de l'ensemble d'un programme en propageant les modifications d'une source réactive (modification d'une variable, entrée utilisateur, etc.) aux éléments dépendants de cette source. Ainsi OpenMusic se doit d'être plus performant pour répondre aux nouvelles attentes.

Suite à ces travaux, le contenu des partitions obtenues via OM devient dynamique. Remarquons que ces partitions sont donc des processus dynamiques. Leur lecture doit toujours respecter un ensemble de règles pour réaliser leur rendu sonore sans pour autant figer leur contenu. Ces partitions en cours de lecture se traduisent par l'exécution de séquences d'actions datées à laquelle s'entremêlent des calculs venant à la modifier. Ces calculs sporadiques dont on ignore la durée ne permettent donc pas d'anticiper l'ordonnancement des tâches pour assurer un rendu déterministe.

Le développement de OM est réalisé via LispWorks [1]. Il s'agit d'un environnement de développement pour langage Common Lisp respectant les normes

1. http ://www.lispworks.com/documentation/lw70/LW/html/lw.htm

CLOS (Common Lisp Object Système). De plus le travail est effectué dans l'environnement d'une machine "grand public". Il y a différentes couches dans un système, au niveau logiciel (la couche la plus haute), le contrôle devient de plus en plus inaccessible vers les couches les plus basses. Ces couches influent sur le comportement global du système tout en étant difficile à observer. Voici un résumé présentant ces couches sous forme de niveau :

— Niveau 1 : Nous pouvons gérer notre propre notion de tâche (appelé threads dans la suite) grâce à notre propre ordonnancement,
— Niveau 1.5 : Nous n'avons pas de contrôle ni sur l'ordonnancement ni sur le ramasse-miettes [2] de LispWorks. Le travail s'étant fait sur un ordinateur 64-bit nous avons fait les tests sur l'environnement LispWorks 64-bit. Le ramasse-miettes de cet environnement semble bien optimisé (ce niveau n'est pas présent pour tous les environnements logiciels, et est dû à l'environnement de programmation LispWorks) [3],
— Niveau 2 : Nous n'avons pas de contrôle sur les threads système (threads noyau),
— Niveau 3 : Nous n'avons pas de contrôle sur les mécanismes de cache et de "context switch" entre processeurs.

L'objectif du stage est de proposer des pistes de réflexion sur le contrôle adaptatif d'un tel logiciel. L'utilisation d'un outil d'analyse produisant des statistiques est donc requis. Ces statistiques permettent d'analyser l'exécution de tâches faites par un système informatique. Dans le cadre d'OM nous avons de plus développé un nouveau moteur d'exécution à groupe d'unité d'exécution (appelé dans la suite moteur multithread) remplaçant un moteur à une unité d'exécution (appelé dans la suite moteur monothread).

In fine, cela permettra d'augmenter la faisabilité de partition interactive exécutée en temps réel. En résumé, une analyse de ces statistiques permettra d'adapter le contexte d'exécution en fonction de la charge de travail du système.

2. Gestionnaire de mémoire qui gère automatiquement la libération de mémoire déjà utilisé, ne servant plus au programme aussi appelé garbage collector.

3. À propos de la gestion du garbage collector dans l'environnement 64-bit voir http ://www.lispworks.com/documentation/lw70/LW/html/lw-74.htm#pgfId-886617.

3 Informatique musicale et média temporel

3.1 Présentation des médias temporels

L'étude réalisée au cours de ce stage porte sur les médias temporels. Il s'agit par exemple de buffers audio potentiellement interactifs. Nous parlerons de média temporel en gardant la notion d'interactivité implicite.

Un média temporel exige un rendu de l'information. Ce rendu peut se traduire par une lecture du média se faisant en temps réel. Cette lecture n'est pas dissociable de l'environnement où elle est effectuée et doit coexister avec d'autres processus présents dans cet environnement. La "mise en temps" du temps différé (calcul des compositions ou productions) vers le temps réel (lecture des compositions et productions) peut donc être perturbée. Le taux de charge de travail d'un ordinateur influe sur la qualité du rendu de ces partitions.

Les médias temporels sont omniprésents dans la CAO. L'informatique musicale dans son ensemble utilise les médias temporels que ce soit dans la composition ou la performance. Actuellement, les systèmes informatiques musicaux sont amenés à travailler dans un contexte où la composition et la performance s'entremêlent. Les représentations de données (par exemple la partition) et les systèmes (par exemple un système de suivi de partition) sont conduits à évoluer dans ce sens.

3.2 Média temporel et interaction

L'interaction avec les médias temporels demande une fragmentation entre le temps différé et temps réel. En effet les processus compositionnels normalement en temps différé doivent s'intégrer dans un contexte temps réel. Ainsi l'ordinateur se retrouve à effectuer du calcul en même temps que la lecture des médias rendue par le calcul préalable. Par conséquent si un calcul ayant lieu pendant la lecture prend trop de temps son résultat sera rendu en retard.

N'utilisant pas de machine dédiée [4] totalement à ce type de travail, nous nous retrouvons dans un contexte Best-Effort (la machine doit faire de son mieux pour assurer que les résultats aient lieu avant une date butoir ou deadline). Lors de la "mise en temps" d'une composition de nombreux processus divers sont présents ce qui les rend difficiles à gérer. En effet le contexte de lecture d'une tâche est perturbé par d'autres tâches qui sont prioritaires au sein du système.

Pour se rendre compte des difficultés de la réalisation de ces interactions nous proposons une échelle en degrés d'interactivité. En fonction du degré, la

4. Les ordinateurs "grand public" ne permettent pas de maîtriser l'allocation des ressources pour nos tâches.

gestion de l'interaction est de plus en plus complexe dans les contextes temps réel :

— degré d'interactivité nul ou quasi nul : par exemple le fait de cliquer sur un bouton "lecture" ou "pause" d'un logiciel envoie à la machine un signal. La machine démarre ou stoppe le rendu du média sélectionné,

— degré d'interactivité moyen : par exemple un jeu video, qui peut être modélisé comme une suite d'actions et de réactions du joueur et de la machine,

— degré d'interactivité total : par exemple les jeux vidéo transmédia qui se déroulent à la fois dans l'environnement réel du joueur (par exemple avec la géolocalisation) et un environnement virtuel (un écran) où des instructions sont données au joueur. Ainsi, celui-ci évolue à la fois dans le monde physique et le monde virtuel.

Dans un contexte plus musical, on rencontre ces degrés d'interactions avec la musique mixte [5] :

— degré d'interaction nul ou quasi nul : un musicien écoute une bande musicale sur une machine. Ici il n'y a donc aucune interaction entre les deux parties,

— degré d'interaction moyen : ici, le musicien interagit avec la bande en pouvant la modifier grâce à des effets lancés via un controleur.

— degré d'interaction total : la vitesse de lecture de la bande suit le jeu du musicien (comme dans Antescofo). Avec l'improvisation homme-machine la machine et le musicien se répondent mutuellement en improvisant.

3.3 Informatique musicale : de la programmation vers l'interaction

3.3.1 À propos de l'informatique musicale

L'informatique musicale est par nature multimédia car on y manipule des données hétérogènes (MIDI [19], fichier audio, vidéo). Les logiciels d'informatique musicale peuvent être catégorisés comme il suit :

.

— les logiciels/langages de notation (Finale, Sibelius, etc.),

5. La musique mixte est une musique écrite nécessitant des interprètes et des sons préenregistrés, sur bande magnétique dans les année 80 et désormais dans des fichiers audio numériques.

— les séquenceurs (par exemple Logic Pro, Live, etc.),

— les logiciels de génération et traitement du signal (plugins, langages, etc.),

— les logiciels de synthèse et composition musicale (Formes),

— les logiciels temps réel (par exemple Max, Pure Data, etc.),

— dans le cadre du stage : les logiciels de CAO (OpenMusic, PWGL, etc.).

3.3.2 Historique

Depuis la genèse de l'informatique, la musique a toujours été un des moteurs de son développement grâce à la composition et à la performance. Ce "moteur" s'incarne par des générations de logiciels et une augmentation de leurs possibilités dans les domaines : de la programmation objet, de l'interactivité, des protocoles de communications, de la notation symbolique, de la représentation de partition et de la synthèse sonore. Ce court historique a pour but de le montrer.

HILLER, en 1956, avec l'ordinateur ILLIAC 1 livre la première oeuvre générée via la CAO : *Illiac Suite for String Quartet*. Par des règles de filtrage stochastique, l'ordinateur accepte ou rejette des pitchs et des rythmes générés aléatoirements [29]. L'utilisateur se contente de fixer les paramètres du programme, que l'ordinateur exécute pour générer la composition.

En 1957 est créé le premier environnement de programmation pour la synthèse sonore **MUSIC I**, suivi par ses descendants **MUSIC N**. Ces langages intègrent déjà le principe de la séparation entre l'orchestre [6] (oscillateurs) et la partition (programme) [36].

Formes [26] est un logiciel créé à l'IRCAM en 1982. Il appartient à la catégorie synthèse et composition musicale. Ce langage utilise le paradigme objet. Il est orienté vers la composition et l'ordonnancement temporel d'objets. Un processus est affecté à chaque calcul à exécuter tout en pouvant échanger des informations avec les autres processus.

Crime a été développé en 1985 par Gérard Assayag et le compositeur Claudy Malherbe. C'est un environnement de création de modèles musicaux formels dont le but est de permettre la traduction des notations symboliques musicales (processus) vers la représentation sous forme de notation (partition).

6. Les instruments créés par des oscillateurs (UGen qui permet d'amplifier, de filtrer et de générer des enveloppes).

Après cela, en 1986, Miller S. Puckette développe le premier langage temps réel musical **MAX/MSP** [24] avec un environnement graphique de patching dédié à la composition et à la performance. Il cherche à être de plus en plus précis sur la granularité temporelle pour répondre le mieux possible aux demandes du temps réel.

Dans le même temps **Csound** [34] est développé, héritier direct de **Music N**. Il a lui aussi participé au développement du paradigme objet dans les langages musicaux et au développement de concepts de synthèse sonore (orchestre).

En 1991, développé par Taube, le langage **Common Music** [32] devient la référence dans les langages de CAO. Il permet d'écrire des relations de haut niveau par la description du son. Il s'agit d'une boîte à outils pour la CAO extensible.

SuperCollider [18] est un langage de programmation de synthèse audio qui redéfinit le genre en 1996. Il garde les notions de générateur de traitement du signal (audio et contrôle) comme MUSIC N mais entremêle la synthèse audio et la partition (les événements musicaux). Cela permet au code d'être encore plus expressif. De plus il embarque un paradigme objet, qui permet au développeur non seulement d'écrire des programmes de synthèse plus facilement, mais aussi de construire des systèmes interactifs pour la synthèse sonore, la composition algorithmique et pour la recherche en informatique musicale.

Ensuite vient l'ensemble des langages de CAO modernes (**Patchwork** [2] entre 1989-1995, **OpenMusic** depuis 1998, **PWGL** [15] depuis 2002) qui sont des environnements de programmation visuels dont nous parlerons dans la section 3.3.4.

Chuck [36] est un langage développé depuis 2002, se prétendant dans la lignée indirecte de **MUSIC-N**. Il permet le contrôle du flux audio avec un modèle de programmation concurrente. Il peut de plus modifier du code à la volée. Cela permet de générer ou interagir avec de la musique interactive en temps réel. Ce langage est essentiellement utilisé pour faire du "live-coding". Sa particularité est d'effectuer les traitements du contrôle et de l'audio à la même granularité : celle de l'échantillon. Ainsi le langage se définit comme "strongly timed".

Avec le protocole OSC [38] utilisé dans **Max/MSP** et **Chuck** les orchestres d'ordinateurs fonctionnent et communiquent plus facilement [33]. Le tout en mêlant le modèle de programmation orienté agent dans les systèmes multi-agent (ce modèle ressemble à la programmation objet mais possède des entités adaptatives, autonomes capables de communiquer facilement entre elles et d'organiser leurs actions).

3.3.3 Dualité temps différé et temps réel

Remarquons que lorsqu'un compositeur écrit une oeuvre, le temps représenté dans la partition est différent de celui qui s'écoule lors de l'écriture. Le temps écrit (celui de la partition) est ce que l'on appelle le temps différé. Le musicien quant à lui s'approprie le temps différé en lisant la partition. Lorsqu'il joue celle-ci en temps réel, il effectue une mise en temps de ce qui est écrit dans la partition.

3.3.4 Les environnements de programmation visuelle

Les environnements de CAO offrent des outils autour de la représentation musicale (rythme, partition, harmonie) avec des approches calculatoires. Le temps musical s'y représente comme une dimension spatiale, ce qui permet un grand pouvoir d'abstraction. Le pionnier de ce type de logiciel est **Patchwork** [2]. On peut citer **BACH** [1] une bibliothèque de **MAX** permettant de faire de la CAO dans une logique temps réel.

OpenMusic est l'héritier direct du langage **PatchWork**. **PWGL** s'inspire également des bases proposées par **PatchWork**. Ces environnements visuels simplifient et rendent abordable la programmation à l'aide de la notion "patching". Le "patching" consiste à connecter des boîtes entre elles. Ces boîtes peuvent être des opérateurs ou des objets. Le résultat est une vision graphique du programme pour l'utilisateur.

3.3.5 Les langages musicaux temps réel

À la différence des logiciels CAO, les logiciels de temps réel entremêlent l'exécution du programme et les calculs en temps borné. Ces applications sont créées pour une utilisation en concert. C'est pourquoi les programmes doivent réagir continuellement aux données reçues en entrée du système (signaux audio, metronome, événements). La réaction à ces entrées entraine des sorties qui doivent respecter au mieux certaines contraintes temporelles.

Max et **PureData** [25] sont des langages graphiques ayant un flux de données réactif. Ils permettent de traiter le signal en temps réel, ce qui fait de ces logiciels de bons outils pour la performance et la création d'applications interactives. Ceux-ci sont en revanche incapables d'exécuter et de représenter des structures compositionnelles complexes en temps réel de par leur limitation en complexité de calcul. D'autre part, ces logiciels ne possèdent pas de vision à long terme.

3.3.6 Vers une mixité de la CAO et des systèmes temps réel

L'opposition entre la composition et le performance (opposition temps différé et temps réel) au fur et à mesure des progrès techniques, devient de

plus en plus floue. Cela se traduit par de nouveaux champs d'applications comme dans le Jeux Video et dans les systèmes d'improvisation automatique (**Improtek**) [22].

De nouveaux outils se développent pour la programmation live en concert (live-coding) avec **Chuck**. Ce langage utilise des structures complexes de traitement sonore en temps réel. Un autre outil moderne est le suivi de partition avec **Antescofo**.

3.4 Présentation de OpenMusic

OpenMusic (OM) est un logiciel de programmation visuel s'appuyant sur le langage de programmation Common Lisp Object System (CLOS). Ce langage est à la fois fonctionnel et orienté objet.

OM est développé par l'équipe Représentation Musicale (RepMus) de l'IRCAM depuis plus d'une vingtaine d'années. L'interface visuelle se présente en deux vues principales :

— Les patchs [7] qui sont constitués d'objets graphiques musicaux connectés entre eux. Ces connexions représentent les liaisons entre les objets (les dépendances de calculs).

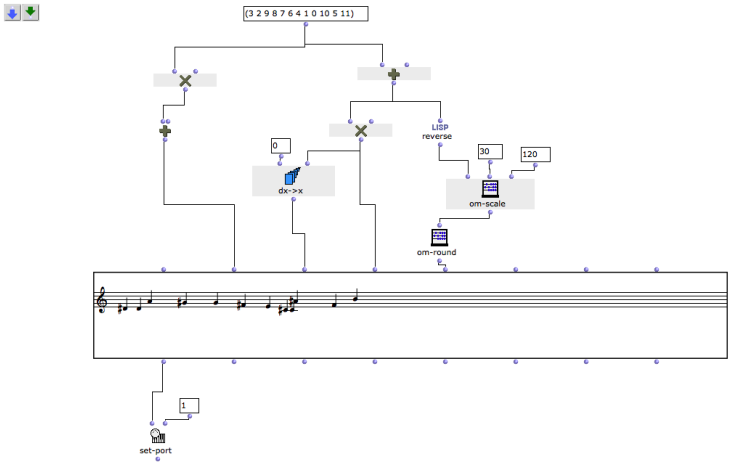

FIGURE 1 – Exemple de patch dans OM 6.9.

7. Programmes visuels.

— Les maquettes [8] se représentent sous forme de patch complété par une timeline [9](voir figure 2). De plus, la création de connexions entre patch à l'intérieur de la maquette permet d'exprimer des contraintes temporelles directement entre objets temporels. On peut mettre en relation ces maquettes d'OM avec les partitions Iscore [17] un séquenceur OSC, qui ne permet que le contrôle.

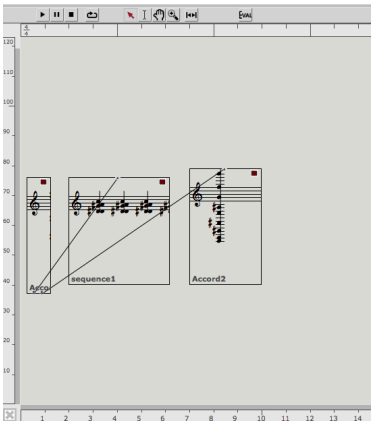

FIGURE 2 – Exemple de maquette de OM 6.9.

3.5 Antescofo un logiciel multimédia interactif

À titre d'exemple de système informatique musical très interactif et nécessitant des ordonnancements dynamiques de tâches devant satisfaire à des relations temporelles complexes, nous décrivons brièvement le logiciel **Antescofo**.

Antescofo (2008) est un logiciel de suivi de partition développé depuis 8 ans par l'équipe Mutant dans l'unité Représentation Musical de l'IRCAM. Il s'agit d'un module logiciel sous forme de bibliothèque pouvant se connecter avec **MAX** ou **PureData**.

Ce logiciel couple un système de suivi de partition avec un langage temps réel réactif et temporisé. **Antescofo** est utilisé principalement dans un contexte de performances mêlant des instruments de musique réels à des dispositifs électroniques de production de sons. On parle de pièces mixtes « électroniques-instrumentales ». L'objectif d'**Antescofo** est de permettre à la partie électronique de réagir en temps réel au jeu du musicien humain, à partir de la détection du tempo et de la position d'un interprète sur une partition

8. Patch contenant des médias temporels et des patchs ordonnés dans le temps.
9. Règle temporelle horizontale permettant de lier la position des objets à une date précise.

pendant qu'il joue.

Grâce à la détection en temps réel du tempo de l'instrumentiste, le compositeur peut écrire les parties électroniques de sa pièce en utilisant des notations temporelles musicales traditionnelles. Différentes stratégies de synchronisation et de rattrapage d'erreurs peuvent être choisies par le compositeur, suivant le contexte musical, pour les actions duratives. Antescofo permet aussi de conditionner des actions à certaines caractéristiques comme l'interprétation (accélération, présence ou non d'une certaine note...), afin que la machine produise un résultat « sous le contrôle de l'interprète », dans le cadre par exemple d'« œuvres ouvertes » ou semi-improvisées. Ainsi, le compositeur peut laisser à l'interprète une certaine liberté, par exemple sur l'ordre d'exécution des passages, des répétitions, etc.

L'implémentation des contraintes temporelles entre les actions spécifiées par un programme Antescofo nécessite des stratégies d'ordonnancement complexes et dynamiques, capables de gérer plusieurs échelles de temps allant de la milliseconde (buffer audio) à l'heure (contrôles à larges échelles pour un opéra par exemple).

4 Ordonnancement

L'ordonnancement se déroule dans un contexte d'exécution qui peut être en temps réel ou en temps différé. Dans ce stage, nous ne cherchons pas à ordonnancer les actions en fonction des ressources disponibles mais à les ordonnancer dans le respect des dates butoirs que nous appellerons deadlines dans la suite. Dans cette partie nous nous concentrerons sur l'ordonnancement dans les systèmes temps réel.

Nous présentons ici quelques principes d'ordonnancement provenant de l'état de l'art sur ce sujet, ainsi qu'une technique d'ordonnancement utilisée dans le contexte du temps différé et le contexte du temps réel que nous lions avec les problèmes d'ordonnancement rencontrées dans la "CAO interactive".

4.1 Principe de l'ordonnancement

Nous avons besoin de formaliser quelques notions d'ordonnancement avant de nous attaquer à la suite. Dans [3] l'ordonnancement est décrit comme deux étapes successives :

— Dans un premier temps le *séquencement* : consiste à construire un plan d'exécution selon la manière de choisir la prochaine tâche,

— Dans un second temps l'*ordonnancement* : consiste à choisir à quel moment envoyer vers un moteur d'exécution une tâche en fonction de sa durée de complétion.

On modélise le traitement des exécutions par des tâches. Une tâche peut être décrite par la loi d'arrivée temporelle séparant deux activations du groupe d'instructions à effectuer.

Soit $\tau = \{\tau_1 ... \tau_n\}$ où τ_i représente une tâche et τ un ensemble de tâches. Il existe trois lois d'arrivée de tâches :

— Tâche à loi d'arrivée périodique : la tâche a une durée constante et une arrivée périodique de valeur T_i. Ce type de tâche est souvent utilisé pour des actions qui doivent être appelées avec régularité. Par exemple la lecture de buffers audio est périodique,

— Tâche à loi d'arrivée sporadique : le temps séparant les deux tâches possède une borne inférieure tel que tempsMin $\geq T_i$. Ce sont des événements irréguliers. Par exemple les tâches de contrôle sur les buffers audio (effets sur les buffers),

— Tâche à loi d'arrivée apériodique : même chose mais sans borne inférieure (voir figure 3).

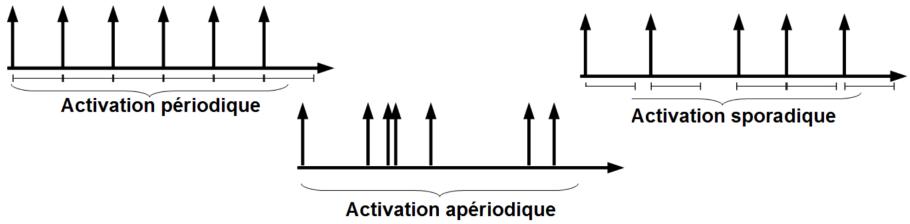

FIGURE 3 – Rendu des lois d'arrivées de tâche.

La durée d'exécution d'une tâche n'étant pas fixe, la pire durée d'exécution de la tâche notée WCET (Worst-Case Execution Time) est aussi un bon indicateur qui permet de faire une méthode de validation de systèmes temps réel. Le WCET à la durée d'exécution la plus pessimiste d'une tâche lors de son exécution par le système.

On mesure le WCET à l'aide d'une analyse statique du code qui analyse le flot d'exécution et les chemins d'exécution à emprunter. Le WCET va de pair avec le meilleur cas d'exécution noté BCET (Best Case Execution Time). Cela permet de créer un étalement des temps d'exécution entre ces deux bornes.

Par contre dans les systèmes "grand public" actuels, le WCET a une très grande valeur [10]. De plus le WCET arrive rarement et donc ne dénote pas d'un phénomène très réaliste dans ces systèmes. Ainsi on préférera de nouvelles techniques de calcul de WCET statistique proposées dans [12].

Il existe deux types d'ordonnancements différents dépendant des modèles d'invocation de l'ordonnanceur :

— *Event Driven* : ce type d'ordonnancement est aussi appelé ordonnancement asynchrone. Il consiste à invoquer l'ordonnanceur lors de la réception d'un nouvel événement [11]. Une technique usuelle est de faire selon la tâche ayant la plus grande priorité (Hightest Priority First ou HPF)

— *Time Driven* : les invocations de l'ordonnanceur sont indépendantes des événements reçus. Ce type d'ordonnanceur utilise l'horloge interne de la machine et peut par exemple demander un ordonnancement périodique (*tick scheduler*) des travaux dans le temps.

Le nombre de tâches n'est pas statique dans un système temps réel . L'ajout de tâches déclenchées par des événements au cours du temps nécessite de faire

10. Cette grande valeur est due à l'apparition de récents processeurs plus complexes, du multicoeurs et de l'utilisation des caches.
11. Par exemple une nouvelle activation de tâche.

de l'ordonnancement dynamique.

Bien que nous ne sachions pas forcément si une action doit avoir lieu, la durée des tâches peut être déterministe. C'est le cas dans les systèmes "*hard real-time*". En ayant au préalable optimisé l'ordonnancement du système en question, il doit absolument assurer le respect des deadlines imposées. Dans l'autre cas, "*soft real-time*" nous ne pouvons pas assurer une durée exécution fixe. Les durées d'exécution suivent des distributions statistiques que nous souhaitons rendre observables par un outil de mesure.

Ainsi dans le contexte temps réel il existe plusieurs types de deadline que nous définissons ci-après :

— *hard deadline* : si les deadlines ne sont pas respectées peuvent entrainer une "catastrophe",
— *soft deadline* : par exemple un rendu audio en retard n'est pas très grave, la machine fait de son mieux pour rendre les résultats au moment voulu (best-effort),
— *firm deadline* qui entraine l'abandon de la tâche si la deadline est outrepassée, cela sans catastrophe, mais diminue considérablement la qualité du rendu. Se référer à la figure 4 provenant de [3].

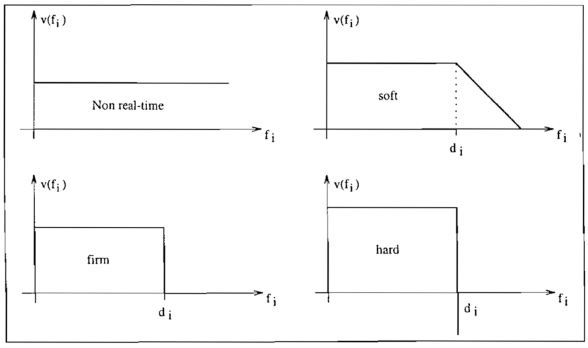

FIGURE 4 – Fonction de coût des différents types de deadlines temps réel.

Il existe de nombreuses métriques pour mesurer l'efficacité des ordonnancements réalisés :

— i représente une tâche donnée,
— C_i représente la date de complétion de i,
— d_i représente la date d'échéance ou deadline de i,
— $L_i := C_i - d_i$ représente le retard ou lateness de i,
— $E_i := \max(0, d_i - C_i)$ représente l'avance ou le earliness de i,
— $T_i := \max(0, C_i - D_i)$ représente la lenteur ou le tardiness de i,

— $D_i := |C_i - d_i|$ représente la déviation absolue de i,
— $S_i := \sqrt{C_i - d_i}$ représente la déviation au carré de i,
— $U_i := \begin{cases} 0 \; si \; C_i \leq D_i \\ 1 \; sinon \end{cases}$ représente la pénalité unitaire de i,
— P_i représente la période d'arrivée de i (cas périodique),
— $Pmin_i$ représente la période minimale d'arrivée de i (cas sporadique),
— O_i représente l'oisiveté entre deux tâches i et i+1 sur le même processeur [12],
— $Tois_{k_i}$ représente l'oisiveté entre deux tâches (i et i+1) au sein d'un même thread k et de la i-ème tâche appartenant à un pool de thread.

Ainsi, on peut raffiner notre définition d'une tâche :
— τ est constitué du triplet $\{C_i, D_i, P_i\}$ dans le cas périodique,
— τ est constitué du triplet $\{C_i, D_i, Pmin_i\}$ dans le cas sporadique.

Les processus (aussi appelés threads) sont des programmes permettant d'exécuter des travaux (fonctions à exécuter). Ces processus peuvent dans les cas les plus simples avoir les états suivants :

— un thread "prêt" est en attente du CPU,
— un thread "bloqué" est en attente d'un événement extérieur,
— un thread "élu" est entrain d'utiliser le CPU.

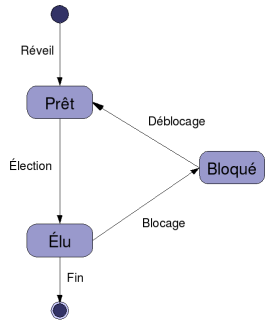

FIGURE 5 – La vie des processus : schéma de la vie d'un thread.

Les systèmes d'exploitation modernes multi-coeurs et multitâches étant préemptifs [13] ont une meilleure réactivité. Par contre en situation de compétition où une tâche possède la priorité sur les autres ils ne sont pas très efficaces car le système continue à partager ses efforts sur l'ensemble des tâches.

Pour simplifier l'étude nous considérerons que la migration (préempter la tâche et transférer son contexte vers un autre coeur) a un coût temporel nul.

12. Appelé "idle time" dans l'état de l'art.
13. Ceux-ci ont la capacité de stopper une tâche planifiée en cours et de la reprendre à son point d'interruption sur le même coeur ou sur un autre s'il y a eu une migration.

Dans ce stage, étant au niveau logiciel nous n'avons pas de contrôle au niveau du noyau qui gère les coeurs de l'environnement.

Dans les cas des systèmes "grand public", les threads permettent l'exécution de tâches au niveau logiciel. Ces threads les transmettent au noyau du système d'exploitation qui les utilise ensuite pour exécuter les travaux à réaliser. Dans le cas des systèmes d'exploitation les plus récents : les threads sont pratiques pour le "multi-tâches" [14].

L'utilisation d'un logiciel/système en temps réel, impose des fluctuations temporelles des durées d'exécution des tâches en raison de :

— la gestion non déterministe des interruptions (préemption),
— un partage équitable entre processeurs gérés au niveau du noyau,
— des micro-exécutions créant des fluctuations temporelles dues au mécanisme de gestion des caches mémoire,
— la gestion de l'horloge interne,
— les optimisations d'utilisation des ressources.

Il existe deux manières d'exécuter un plan dans un contexte d'ordonnancement dynamique :

— *dynamic planning-base approaches* : consiste à regarder au moment de l'exécution si le système temps réel peut assurer le respect des deadlines cela en vérifiant la faisabilité du plan avec un contrôle d'admissibilité des tâches en fonction de leurs deadlines et de leurs priorités,

— *dynamic best-effort approaches* : le système fait de son mieux pour respecter les deadlines rencontrées sans vérifier la faisabilité avant la fin supposée de la tâche.

4.2 Une technique d'ordonnacement : Earliest Deadline First

Dans [31] nous est présenté un ensemble d'algorithmes du type Earliest Deadline First (EDF). Ces algorithmes doivent exécuter en premier la tâche ayant son échéance avant les échéances des autres cela par rapport à l'ensemble des tâches à effectuer.

Dans l'exemple ci-après, nous avons l'arbre de dépendance des tâches (voir figure 6) soumis à un l'algorithme d'ordonnancement EDF dans un cadre statique. Les tâches proposées respectent les dates d'échéances ainsi que les

14. Pouvoir exécuter plusieurs programmes en même temps.

durées présentées dans le tableau suivant :

tâches	durée	deadline
T1	3	3
T2	2	9
T3	4	7
T4	3	13
T5	3	15

Nous présentons une manière graphique de représenter la planification émise par le système avec un diagramme de Gantt, celui-ci permet de représenter la planification des différentes tâches à effectuer (voir figure 7).

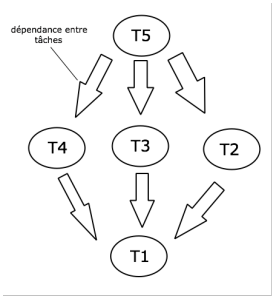

FIGURE 6 – Exemple de diagramme de tâches dépendantes.

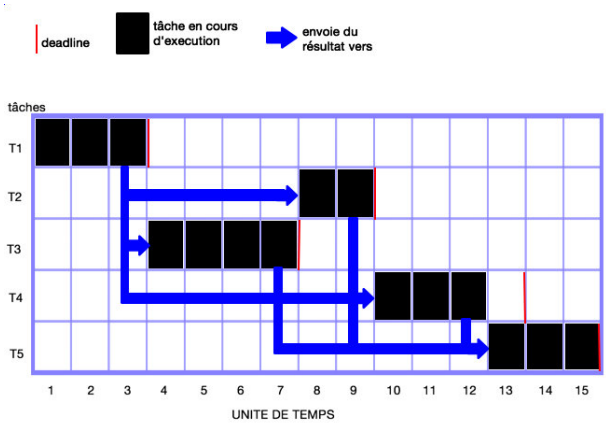

FIGURE 7 – Exemple de diagramme de Gantt pour l'exemple EDF.

Dans le cadre du temps réel, EDF est largement utilisé car il est optimal dans différents cas. L'ajout dynamique de tâche entraine une replanification de la séquence de travaux. Dans ce cas la figure 7 serait identique mais avec une barre de lecture qui en atteignant des instants précis, déclencherait l'arrivée d'événements.

On s'assure que la replanification est correcte via "la règle de Jackson" : dans les cas où il n'y a pas préemption, toute séquence est optimale si les travaux sont ordonnés dans un ordre non décroissant des deadlines.

Dans les cas préemptifs avec appel synchrone périodique de tâche cela se vérifie si et seulement si le facteur d'utilisation du processeur U est inférieur ou égal à 1 tel que $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ [16]

— EDF est optimal pour l'ordonnancement des tâches sporadiques ou périodiques préemptives [23],
— EDF est optimal pour l'ordonnancement de tâches sporadiques ou périodiques non préemptives non concrètes [14],
— l'optimalité n'est plus garantie pour des tâches périodiques concrètes non préemptives [14].

Les algorithmes EDF pour des tâches indépendantes dans un cas d'ordonnancement fixe défini au niveau des tâches (Fixed Priority at Job Level (FPJL)) ont pour but d'assurer la faisabilité des tâches ainsi que leur optimalité dans les cas de test sur des matériaux monoprocesseurs. De plus en cas de surcharge du système, l'algorithme EDF est soumis à un "effet avalanche". Les échéances ne sont pas respectées en raison du retard cumulé.

4.3 Problématique de l'ordonnancement dans les logiciels multimédia modernes de type CAO

Les logiciels multimédia modernes doivent souvent jouer à la fois sur les échelles du temps réel et du temps différé. C'est le cas de la CAO où les nouvelles techniques d'ordonnancement des tâches qui lui sont transmises sont à développer. En fonction des événements reçus par le système, des mises à jour du plan [15] sont nécessaires et demandent une phase de ré-ordonnancement [16].

Dans [21] un exemple proposé d'ordonnancement dynamique est la modification de la séquence d'accords faite par ImproteK dans le futur, selon des réactions à des changements de paramètres extérieurs. Pour le système d'ordonnancement cela se traduira par des changements à faire en ligne de la suite d'accords. Il modifie son plan sur la prochaine fenêtre de temps (voir figure 8 inspiré de [21]).

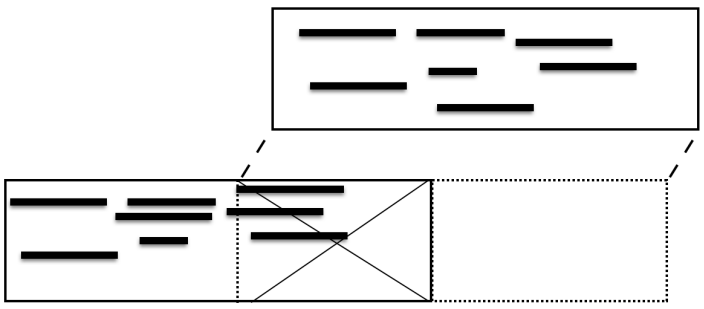

FIGURE 8 – Rendu de changement dynamique de suite d'accords.

15. Séquence d'opération ordonnée dans le temps à exécuter.
16. Devoir reconstruire un plan d'exécution.

5 Environnement du stage et développement

Cette partie a pour objectif de présenter l'environnement de programmation du stage et les travaux réalisés au sein d'OpenMusic 7 (OM7). La description du logiciel OpenMusic dans 3.4 concerne OpenMusic 6.9 et ses ancêtres. OM7 est réactif, nous le présentons ci-après.

5.1 Vers un logiciel de CAO réactif

Les avancées récentes dans le développement de OpenMusic ont introduit des capacités réactives pour le langage visuel [10] [8]. Ces développements s'intègrent dans une optique d'homogéinisation de l'ensemble des travaux de la CAO au cours du projet EFFICAC(e) [9].

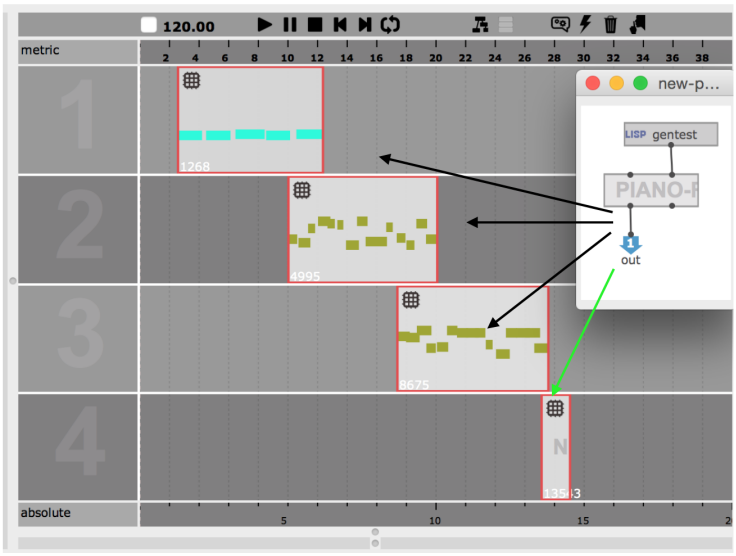

FIGURE 9 – Aperçu de l'environnement OM7 : Maquette, Patches générant des séquences MIDI. Dans cet exemple le dernier patch n'a pas encore été calculé.

Le concept de réactif va de pair avec la notion d'événement. Lorsqu'un nouvel événement arrive le système doit réagir. Un nouveau système d'ordonnancement dynamique dans OM7 a été développé [7]. En effet, OM joue désormais à la fois dans le contexte hors-ligne (composition) et dans le contexte en-ligne (performance) en mêlant leurs techniques de traitement des données. Les processus musicaux de composition et de performance s'unissent à présent dans un même cadre, toutes interactions deviennent partie intégrante de ce cadre. Qu'ils soient génératifs pour la composition ou transformatifs pour la performance.

Un processus compositionnel dans un système se divise en trois étapes :

— Le calcul (par le “moteur de calcul”) consiste à transformer une séquence d’instructions en objets musicaux,

— L’ordonnancement (par l’ordonnanceur) consiste à transformer les objets musicaux en séquences temporelles d’actions appelées plan,

— La restitution (par le répartiteur) consiste à déclencher les actions du plan en temps voulu (voir figure 10).

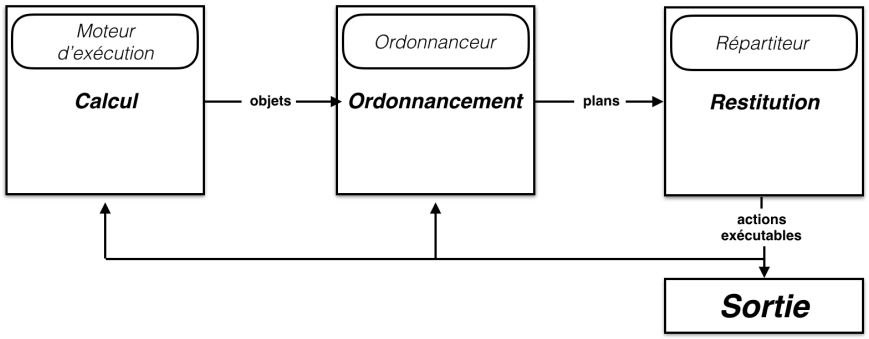

FIGURE 10 – Fonctionnement de l’ordonnancement dans OM7.

L’ordonnancement est ici “demand-driven” et peut réordonnancer le plan, ce qui est nécessaire car les données musicales peuvent réagir à un événement. Notons que ces événements sont gérés uniformément, qu’ils proviennent de l’environnement extérieur ou du système lui même.

L’outil de mesure créé au cours de ce stage a pour but d’aider la machine à produire des résultats en temps et en heure voulue en l’adaptant au contexte d’utilisation. Cela est possible par l’observation du temps de calcul d’une même tâche appelée périodiquement, par exemple la tâche de lecture des buffers audio utilisés dans un logiciel. Dans cette optique nous pouvons adapter l’horizon [17] à la charge de travaux demandés.

5.2 Statistiques sur l’environnement OM

Afin d’avoir plus d’informations sur le déroulement des programmes dans le système nous avons implémenté une API de statistique des temps d’exécution.

17. Jusqu’où regarder dans le futur pour produire un plan d’exécution.

Ainsi, nous récupérerons le temps de commencement d'une tâche, le temps de fin d'une tâche et l'instant où est censé commencer la tâche. Grâce à ces mesures, nous pourrons récupérer le temps d'exécution d'une tâche ($debut - fin$) et le retard d'une tâche ($debut - debutSuppose$). Dans la même lignée nous récupérons les temps d'exécution des opérations de l'ordonnanceur.

En définitive, nous récupérons la moyenne de la durée d'une tâche, sa médiane, sa variance, son écart max (appelé aussi range) et à partir de cela un histogramme de distribution.

- La moyenne se définit par $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$,
- la médiane satisfait l'égalité : $\mathrm{P}(X \leq m) \geq \frac{1}{2}$ et $\mathrm{P}(X \geq m) \geq \frac{1}{2}$,
- la variance est définie par $\mathrm{Var}(X) \equiv V(X) \stackrel{\text{def}}{=} \mathbb{E}\left[(X - \mathbb{E}[X])^2\right]$. ,
- l'écart type se caractérise par $\sigma_X = \sqrt{\mathrm{Var}(X)}$.

Voici les étapes du "protocole" de prise d'informations et d'affichage pour chaque exécution.

- I : La prise d'information se fait à partir d'une maquette d'OM. Dès que la lecture est lancée les données sont intégrées en temps réel (mise à part la fréquence mise à jour a posteriori) dans une structure de données.
- II : On récupère ensuite les données qui nous intéressent. Puis nous les enregistrons dans des fichiers par types de données.
- III : Enfin on affiche les résultats automatiquement via une commande ouvrant les fichiers sur Gnuplot [a] sous forme de graphes et d'histogrammes

a. Traceur permettant d'afficher des fonctions sur un graphe 2D ou 3D..

5.3 D'un moteur de calculs monothread vers un moteur multithread

Dans l'ancienne architecture d'OM le moteur de calcul était monothread. Nous avons décidé durant ce stage de réaliser un pool de threads fonctionnant de la façon décrite dans la figure 11.

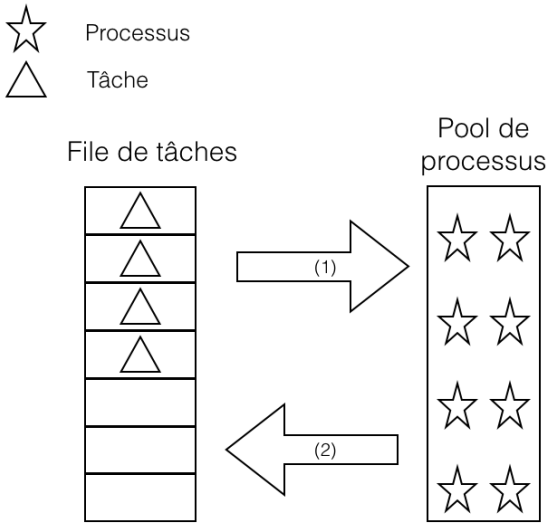

FIGURE 11 – Fonctionnement du pool de threads.

À l'étape (1), nous demandons au thread de prendre les tâches. Dès qu'une nouvelle tâche est envoyée dans la file de tâche, on signale à tous les threads qu'ils peuvent se servir. L'étape (2) consiste à exécuter le calcul demandé et envoyer le résultat en sortie.

5.4 Sauvegarde des données récoltées

Suite à l'exécution d'une maquette, nous enregistrons dans des fichiers textes les données récoltées (temps d'exécution, retard, etc.). Ces données sont triées en fonction d'un thread précis ou du nombre de threads actifs au moment du calcul. Le nombre de threads actifs permet de se rendre compte de la charge de travail demandée. En effet, le nombre de threads actifs représente le nombre de threads en cours d'exécution dans les processeurs. Avec un MacBook Pro (mi-2015) possédant un processeur Intel Core i7, nous pouvons observer :

— de 1 à 4 thread : occupation des coeurs physiques de la machine,
— de 3 à 8 thread : distribution de plus en plus élevée dans les coeurs logiques (avec l'hyperthreading [18]),
— 8 et + : saturation en cas de surcharge [19] de l'ensemble des coeurs logiques. Se référer à la figure 12.

Nous visualisons les résultats via Gnuplot pour obtenir des graphes bidimensionnels et tridimensionnels où nous pouvons effectuer des observations.

18. Un coeur physique de la machine crée deux processeurs logiques agissants comme un coeur (propres registres, préemption individuelle.)
19. Il n'y a pas d'oisiveté du processeur.

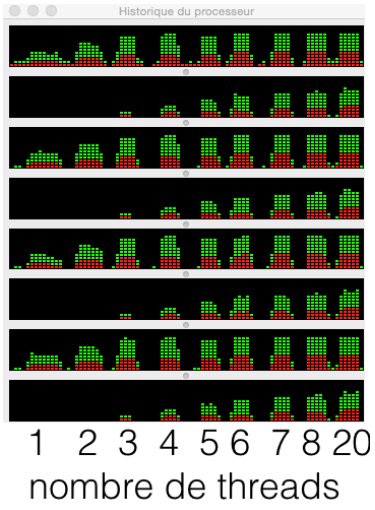

FIGURE 12 – Activité des processeurs en fonction du nombre de threads occupés pour une même tâche de calcul.

Ces données récoltées peuvent être pondérées par la fréquence du CPU obtenue via Intel(R) Power gadget. L'outil d'Intel permet d'observer le comportement de l'horloge interne de la machine. En effet pour des raisons de performance de la machine et du matériel, les processeurs ne tournent pas à plein régime en permanence. Par conséquent la fréquence du CPU oscille et permet d'éviter la surchauffe de la machine, se référer à la figure 13.

Nous analysons trois types d'exécutions utiles dans la CAO et qui demandent des actions différentes au système :

— l'exécution et le calcul d'opérations simples unitaires qui modifie des zones mémoire,

— l'allocation mémoire qui demande au système de réserver de la place dans la mémoire,

— les calculs et la lecture de buffers audio qui doivent chercher en mémoire les données afin de les lire.

Ces observations permettent de mieux comprendre le comportement du système en fonction du contexte où il se trouve. L'opération de ré-ordonnancement du plan en fonction du contexte construira des "plans adaptatifs" . Par conséquent les observations de statistiques concernant les types d'exécutions seront utiles à la création de plans adaptatifs. De plus, en fonction de l'horizon choisi, le nombre d'actions récupérées par le répartiteur devient plus important ou plus

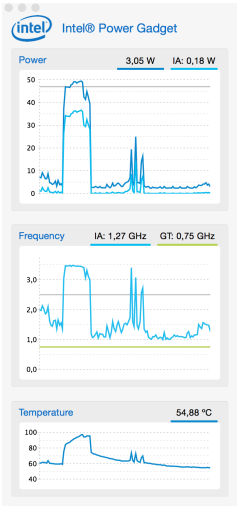

FIGURE 13 – Vue de l'interface graphique d'Intel(R) Power gadget qui permet d'observer le changement de fréquence des coeurs. Dans le travail nous utilisons le fichier de log qui permet de tracer ces changements.

faible. L'horizon est la distance temporelle d'acquisition d'actions par rapport à la barre de lecture dans le plan.

5.5 Gestion des dépendances entre tâches

Nous avons ensuite généralisé cette construction à la forme de "directed acycled graph" (DAG) pour permettre des interdépendances entre branches, ce qui n'est pas possible avec des arbres (voir figure 14).

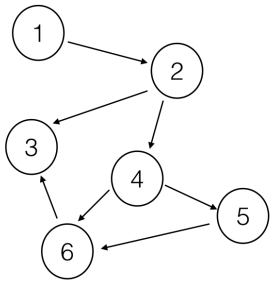

FIGURE 14 – Représentation d'arbre de dépendance k-aire en forme binaire et DAG.

Cette structure a été réalisée dans le but de pouvoir représenter les tâches dépendantes au sein d'OM. Un groupe des boîtes connectées entre elles équivaut à une telle structure. Ces connexions montrent la dépendance d'une boîte dont l'entrée dépend du résultat de la sortie de l'autre.

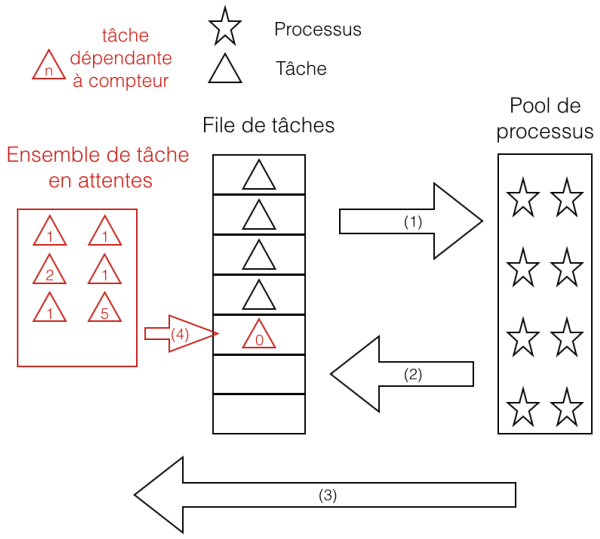

FIGURE 15 – Gestion de tâches dépendantes dans le moteur d'exécution d'OM.

Dans la figure 15, les étapes (1) et (2) sont les mêmes que dans la figure 11, l'étape (3) signifie qu'une fois le calcul de la tâche terminé, on décrémente le compteur du/des parent(s) de la tâche qui vient d'être exécutée. Une fois que le compteur d'une tâche en attente est à zéro, on passe à l'étape (4) qui consiste à mettre la tâche dans la file de tâches. Si elle aussi elle a un/des père(s) elle devra le(s) décrémenter.

6 Résultats par l'observation

En effectuant des observations sur le "Niveau 1" des couches de contrôle du système (cf partie 2), nous espérons pouvoir faire des déductions sur le comportement des niveaux supérieurs dans le but de permettre l'adaptativité du système notamment dans des contextes de surcharge de travail ou perturbation du système. Dans cette partie nous émettons un certain nombre d'hypothèses que l'on espère retrouver par l'observation dans des cas pratiques.

6.1 Notions pour l'observation

- le nombre de threads actifs est le nombre de threads occupés au cours d'une exécution,
- les tâches dépendantes sont représentées par un DAG,
- les tâches en attentes sont des tâches dépendantes contenues dans une structure DAG. Ces tâches attendent que leurs fils finissent d'être exécutés, afin que leur compteur soit décrémenté jusqu'à zéro pour être exécutées à leur tour,
- l'epsilon pourcentage permet de calculer le pourcentage de variation par rapport à un temps nominal d'une tâche,
- la fréquence dont on parlera désigne la fréquence à laquelle tourne le CPU à un moment donné,
- la règle des 10% : la distribution des durées des exécutions des tâches varie majoritairement autour de la moyenne des durées des exécutions plus ou moins 10%. Nous avons la majorité des temps relevés dans cette "tranche",
- un logiciel est dans un contexte de surcharge si l'oisiveté $Tois_{k_i}$ (se référer à 4.1) de tous les threads k du pool de threads pour toutes les tâches i d'un groupe de tâches sont nul. Autrement dit, le contexte de surcharge d'un logiciel a lieu quand un groupe de tâches est calculé sans interruption,
- un environnement multiprocesseur est en situation de surcharge si l'oisiveté des processeurs est proche de 0 pendant une certaine durée. On calcule l'oisiveté total en pourcentage avec : $O_{total} = 100 - C_{total}$ où C_{total} est le taux de charge en pourcentage qui équivaut à la charge de l'ensemble des processeurs.

6.2 Observations

6.2.1 Libération de mémoire périodique dans LispWorks

Plusieurs conclusions ont été possibles à partir de l'observation des temps d'exécution. Par exemple, sur un long test de fonction allouant de la mémoire, on peut s'apercevoir du passage du ramasse-miettes (garbage collector) de LispWorks . On observe que le temps d'exécution augmente soudainement de façon périodique. En effet l'allocation mémoire marche par générations de segments

mémoires [20]. Pour résumé, le passage périodique du ramasse-miettes influe sur la durée des exécutions.

6.2.2 Nombre de threads optimal dans le pool de threads

Sur la figure 16, on peut observer qu'entre 4 et 8 threads, nous sommes à une vitesse optimale. On utilisera plutôt 8 threads en vue de prévenir certains cas où des threads peuvent être bloqués trop longtemps et se retrouve "moins productifs". Par exemple dans les cas :

— d'une attente d'un appel système,
— ou d'une attente pour retrouver un élément dans la mémoire. [21]

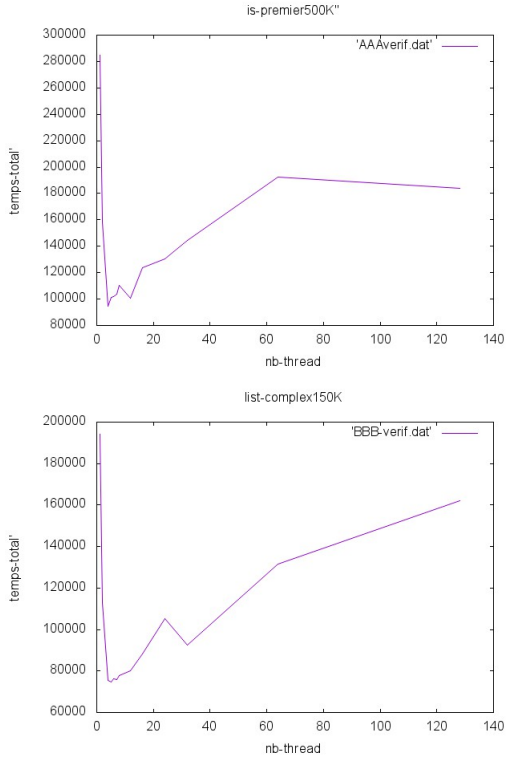

FIGURE 16 – Une fonction représentant le temps total d'exécution d'une maquette avec une fonction allouant de la mémoire et une fonction calculatoire en fonction du nombre de threads du pool.

20. Voir : http ://www.lispworks.com/documentation/lw70/LW/html/lw-78.htm#pgfId-889356
21. Parcours de mémoire hiérarchique, sur plusieurs pages mémoire, ce qui peut être très long.

Dans le cadre de l'adaptation dynamique ces résultats nous intéressent. Dans le pool de threads, il est possible d'ajouter ou de retirer dynamiquement des threads. Cela peut s'avérer pratique dans les cas de transition entre un cas de surcharge de tâches et un cas de non surcharge.

6.2.3 La variation dépend de la fréquence

Au cours du stage nous avons pu établir une supposition importante : la dispersion des temps d'exécution est dépendante de la fréquence du CPU au moment de l'exécution. Ainsi si nous pondérons les temps d'exécution par fréquences au sein de cette exécution. Le résultat est la diminution de la variance par rapport au cas classique.

6.3 Étude de cas : tâche de calcul à travers les statistiques

Nous donnons ici un panel de statistiques et d'interprétations à propos d'une tâche de calcul [22]. On comparera deux cas, le cas où seul le programme demande de la puissance de calcul du CPU, et le cas où on sature les bus d'informations du système avec un autre logiciel [23], tout en exécutant la maquette. Le nombre de threads présents dans le pool de threads est un paramètre à prendre en compte. Parmi les nombreux fichiers générés, voici une sélection avec des commentaires.

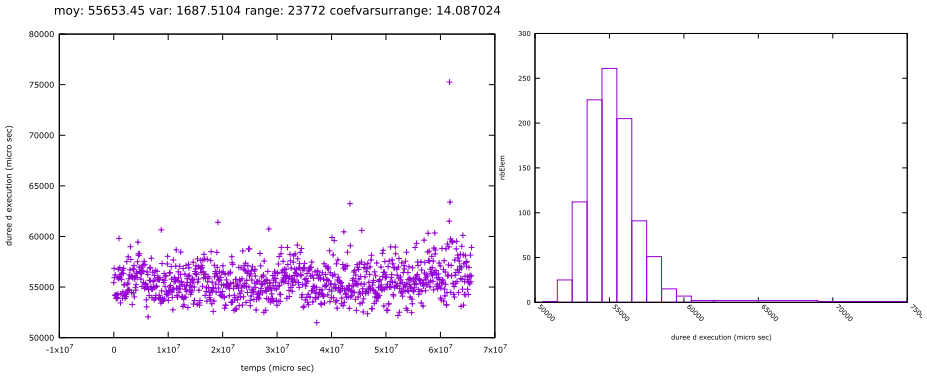

FIGURE 17 – "is-premier", 1 thread, durée d'exécution et son histogramme.

La figure 17 vérifie l'hypothèse des 10%. En effet la fonction de distribution (à une exception près) des exécutions se trouvent entre 50000 et 60000 avec une moyenne de 55000 microseconde (μs) environ (99.5% des valeurs se trouve

22. La tâche de calcul utilisée pendant toute cette étude de cas porte sur la tâche "is-premier" appelée toutes les 20 millisecondes (ms). Cette fonction cherche naïvement si 25000 est premier en passant en revue s'il est divisible ou non par les nombres le précédant.

23. de manière à utiliser un maximum de coeur.

dans cette écart). L'écart max du fait de l'exception entraîne que le coefficient "variance/ecartmax" est assez haut. De plus l'allure de l'histogramme est une distribution gaussienne.

L'étape suivante est donc de vérifier si notre hypothèse quant aux fréquences variant dans le CPU lors de l'exécution est un facteur d'instabilité. Toutes les 20ms (le pas minimal proposé par l'outil intel) les fréquences sont mesurées au cours de l'exécution. On obtient la figure 18.

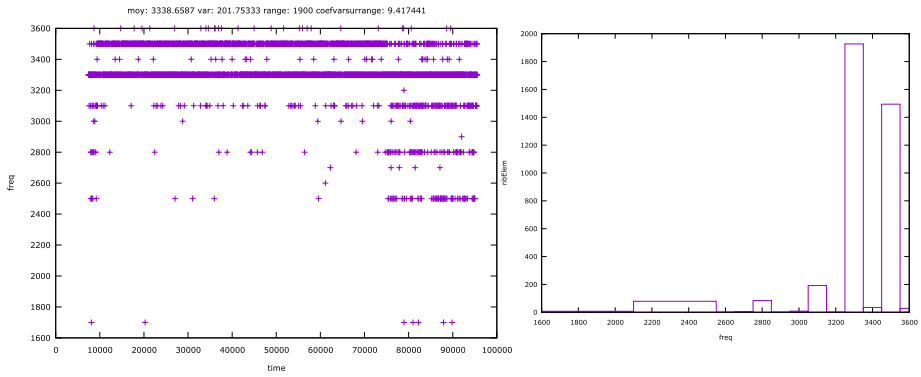

FIGURE 18 – "is-premier", 1 thread, fréquence du CPU lors de son exécution et son histogramme.

On observe donc que la fréquence oscille de quelque giga hertz (Ghz). Grâce à l'outil d'Intel nous avons aussi les dates où ont lieu les mesures de fréquences. En pondérant les temps d'exécution par les fréquences sur chaque sous-partie de la manière décrite dans la figure 19 :

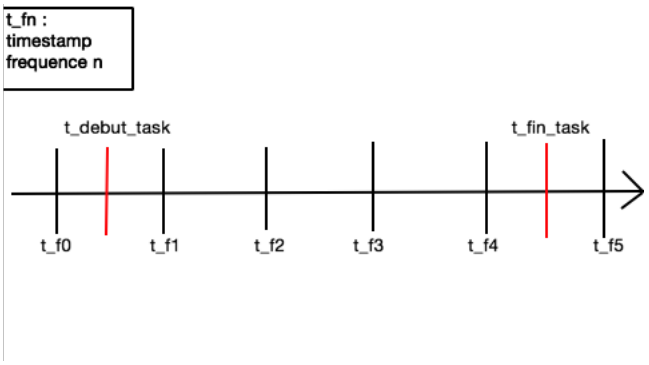

FIGURE 19 – Représentation de l'horodatage de la tâche et prise de valeur de fréquence de l'ordinateur.

En suivant la formule :
$(t_{f1} - t_{debut-task})/f0 + (t_{f2} - t_{f1})/f1 + ... + (t_{fn} - t_{fn-1})/fn - 1$
Le temps d'exécution pondéré ainsi permet d'obtenir la figure 20 :

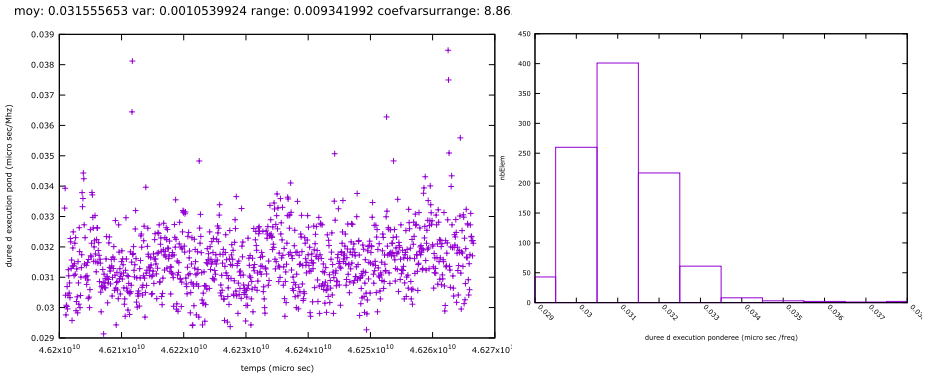

FIGURE 20 – "is-premier", 1 thread, durée d'exécution pondérée par fréquence et son histogramme.

On peut observer que le coefficient "variance/ecartmax" est passé de 14.08 en observant l'exécution normale à 8.86 en observant les temps d'exécution pondérés par les fréquences du CPU. Cela démontre que la fréquence instaure une hausse de l'indéterminisme au sein du système.

Dans la figure 17 sur les exécutions sans pondération, remarquons que la durée d'une exécution dure en moyenne 55ms alors que la fonction est appelée tous les 20ms, ainsi nous devons ajouter un certain nombre de threads pour diminuer le retard (voir figure 21).

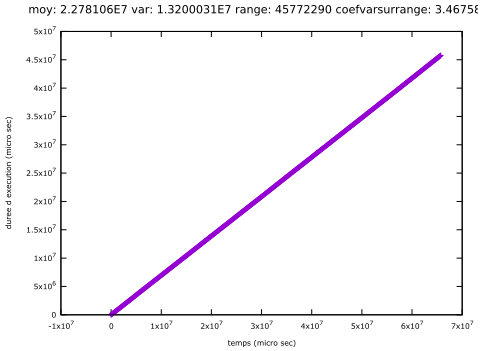

FIGURE 21 – "is-premier", 1 thread, retard cumulé au cours de l'exécution de la maquette.

On observe que le retard cumulé croît de façon linéaire. Pour faire baisser ce retard nous utilisons donc le moteur multithread en lui affectant non pas un mais plusieurs threads (nous avons vu précédemment que le nombre de threads optimal se situe entre 4 et 8 threads)(voir figure 22).

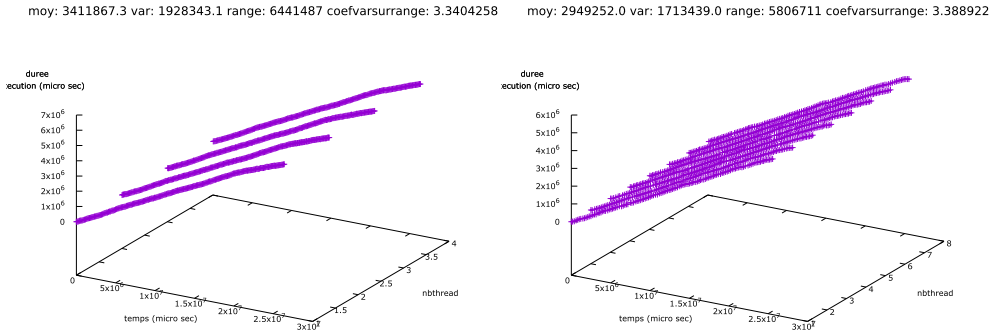

FIGURE 22 – "is-premier", 4 et 8 threads, retard cumulé.

On observe que le retard cumulé croît bien moins vite que dans les cas monothread. De plus les retards cumulés finaux (le dernier point par threads de chaque graphe) paraissent très proches dans les cas "multithreadés".

L'allure des graphiques au niveau du cumul de retard de chaque thread se ressemble. Ainsi on peut conjecturer que l'ordinateur traite équitablement les demandes de chaque thread.

De plus, comme il y a du retard cela implique que les threads sont tous actifs durant l'exécution sauf au début et à la fin (voir figure 23).

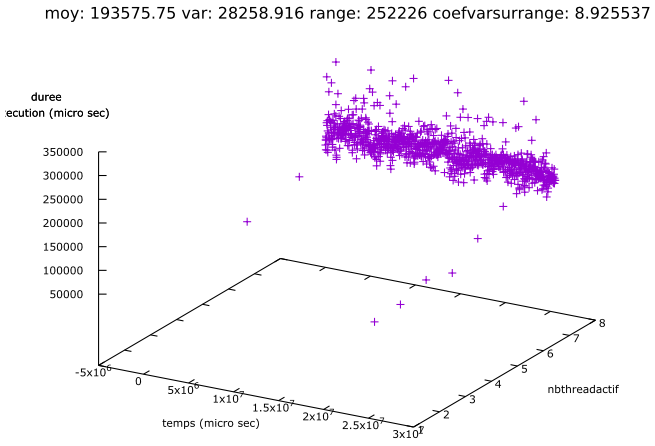

FIGURE 23 – "is-premier", 8 threads, en terme de temps d'exécution par threads actifs.

De plus le temps d'oisiveté des threads doit être faible (voir figure 24).

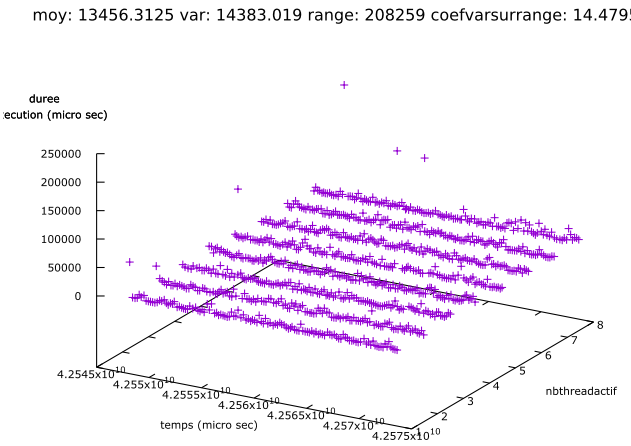

FIGURE 24 – "is-premier", 8 threads, en terme de temps d'oiseveté par threads.

Le temps n'est pas de zéro à cause du temps que prend la mesure et du temps que le thread lorsqu'il choisit une tâche dans la file de tâches.

Dans le cas de perturbation un bon indicateur est de regarder, au niveau logiciel, l'oisiveté de l'ensemble des processeurs (voir figures 25 et 26).

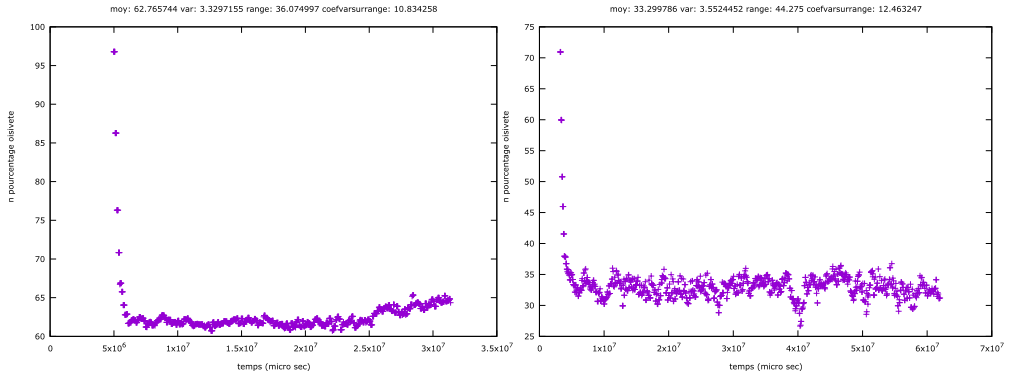

FIGURE 25 – Oisiveté en pourcentage des processeurs en fonction de la charge.

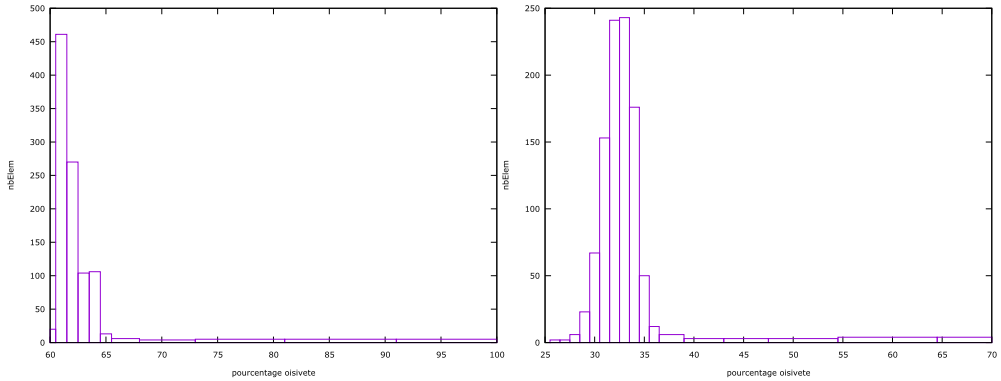

FIGURE 26 – Histogramme des oisivetés en pourcentage des processeurs en fonction de la charge.

On peut voir que l'oisiveté descend à la même allure dans les deux graphes (dans le premier de quasiment 100 à 60 % et dans le deuxième de 70 à 30%). On peut observer aussi que le temps total d'exécution de la maquette dans le cas chargé prend bien plus de temps que dans le cas non chargé.

Nous allons observer à présent les temps d'exécution de ces deux cas (figures 27 et 28).

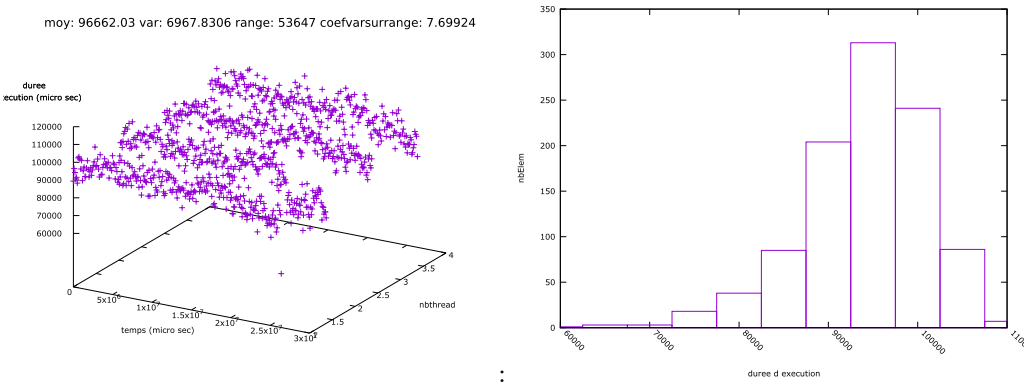

FIGURE 27 – Temps d'exécution is-premier 4T et son histogramme.

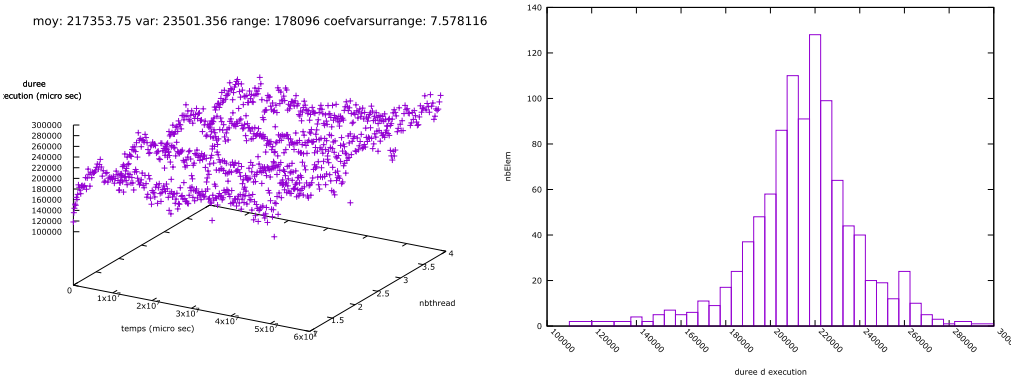

FIGURE 28 – Temps d'exécution is-premier 4 threads et son histogramme en cas de perturbation.

Remarque dans le cas non perturbé :

Le temps de calcul par tâche est plus grand que dans le cas monothread 70000 contre 100000 μs (microseconde) avec 4 threads, mais il faut prendre en compte que nous sommes dans un contexte concurrentiel entre les threads travailleurs, donc la durée d'exécution de la maquette appelant périodiquement une fonction définit par l'utilisateur est inférieur. Mais les threads peuvent traiter les tâches lorsqu'ils sont plusieurs et ont accès à plusieurs coeurs pour travailler en parallèle. Par conséquent le temps total de l'exécution de la maquette est bien inférieur en temps du cas monothread.

Lorsque l'on compare les cas chargés, on peut observer que l'étalement des temps d'exécution lui aussi augmente. Mais on peut espérer une confirmation de la règle des 10 pour cent : si c'est le cas, on peut déduire que la charge de

travail de l'ordinateur influe tel un facteur de dilatation.

Par exemple, dans le cas sans perturbation, on obtient 86.09 % de l'exécution des tâches comprises entre la moyenne + ou - 10 % de la moyenne et avec perturbation : 71.64329 %.

De plus, si on effectue la même méthode sur les fréquences pondérées on obtient : dans les cas sans perturbation et avec perturbation les valeurs en pourcentage de tâche, respectant la règle des 10%, à la baisse.

6.4 Tâche périodique d'accès mémoire

Nous utilisons un outil développé par l'équipe ISMM (interaction son-musique-mouvement). il s'agit ISMM Audio Engine (IAE) est un moteur audio versatile pour la synthèse sonore basée sur le contenu. Ce moteur developpé en C++ a été intégré dans OpenMusic sous forme de librairie dynamique. IAE permet la synthèse granulaire et la synthèse concaténative. L'objet IAE dans OM permet donc de jouer des granules d'un son choisi au préalable. Les granules y sont indexés de 1 à n (n étant le nombre de granules choisi en argument) disposées aléatoirement. L'objet IAE nous intéresse car il alloue de la mémoire pour les granules et les joue sous forme de buffer audio (voir figure 29).

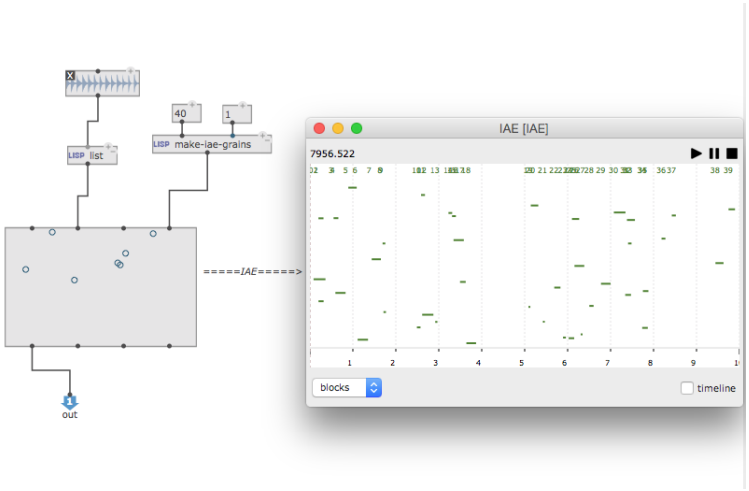

FIGURE 29 – Objet IAE.

La tâche appelée consiste à lire des buffers générés par IAE. Cette tâche crée des buffers audio durant 2 secondes et qui sont appelés toutes les 80ms. Ces buffers audio sont lus en parallèle sur la même sortie. Le recouvrement entre chaque buffers lus suscite une situation équivalente à la lecture d'une

table de mixage 25 canaux.

Avec la règle 10% on observe :

nombre de threads	avec perturbation	sans perturbation
1	94.4%	64%
4	96%	14.86%
8	99%	85.8%
25	98.26087%	16.26%

Ainsi lors d'une lecture de buffer audio, un bon nombre de threads s'occupant de ce cas est donc de 8 (dans le cas sans perturbation 99% des exécutions se trouve autour de 10% de la moyenne (figures 30 et 31).

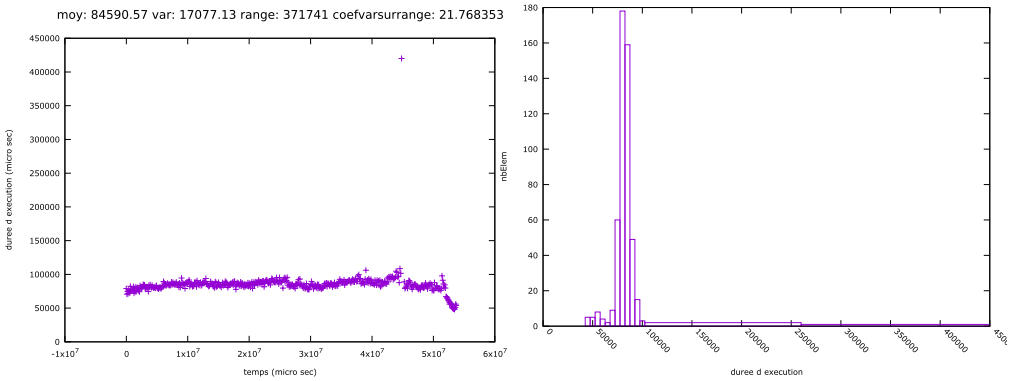

FIGURE 30 – Temps d'exécution IAE 1 thread en environnement perturbé et son histogramme.

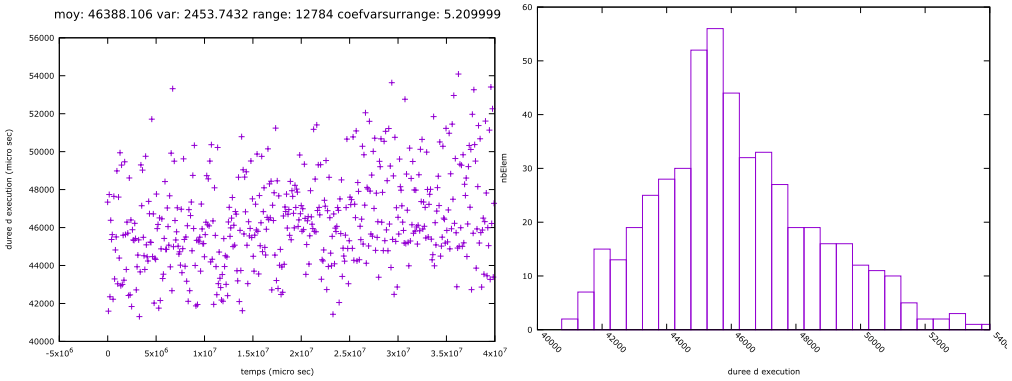

FIGURE 31 – Temps d'exécution IAE 1 thread et son histogramme sans perturbation.

Dans les autres cas lorsque l'exécution est perturbée, la durée d'exécution de la lecture de buffer audio dure plus de temps que l'espace entre chaque appel de buffers. Ce qui se traduit par des bugs ou du son inexistant (figures 32 et 33).

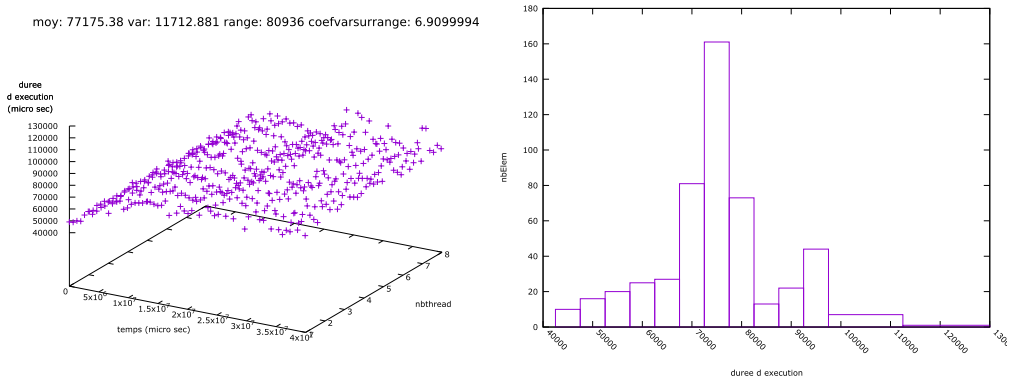

FIGURE 32 – Temps d'exécution IAE, 8 threads, et son histogramme en cas de perturbation.

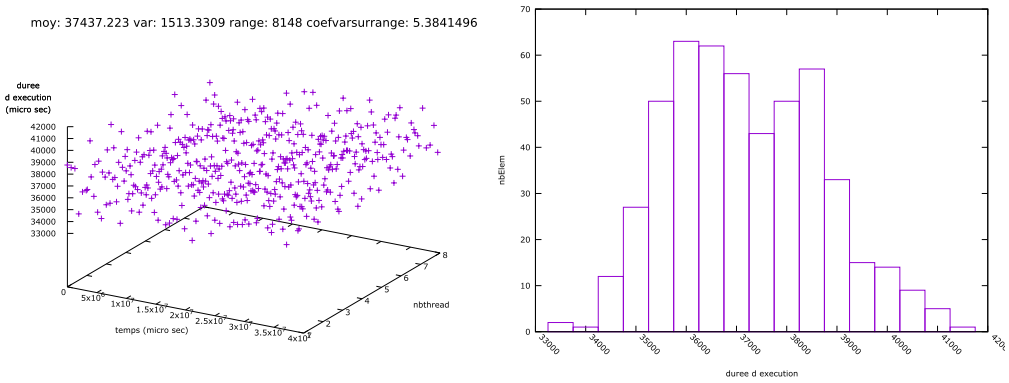

FIGURE 33 – Temps d'exécution IAE, 8 threads, et son histogramme sans perturbation.

Un dernier phénomène intéressant semble être la variation des fréquences du CPU dans les cas perturbés, on peut voir qu'elle décroît un peu comme une fonction en escalier (figures 34 et 35).

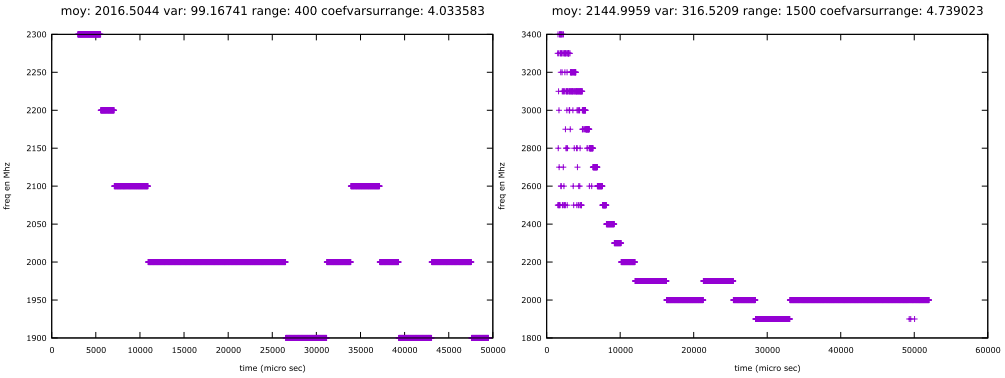

FIGURE 34 – fréquence CPU, 1 et 8 threads, IAE cas perturbé.

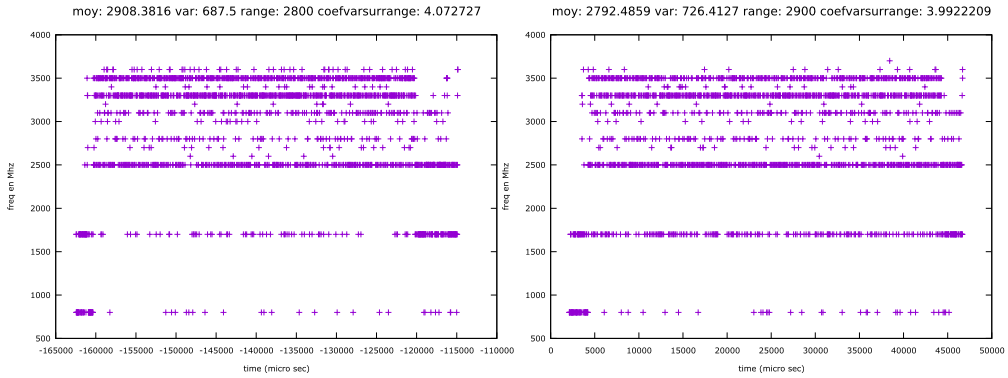

FIGURE 35 – fréquence CPU, 1 et 8 threads, IAE non cas perturbé.

Ce dernier varie beaucoup alors que dans le cas perturbé la fréquence change à une allure plus stable. On peut supposer que cela est dû à l'assignation de régime de fréquence à la tâche perturbant notre tâche d'OM.

7 Conclusion et Perspectives

Bien que présentant des contraintes “temps-réel souple” les logiciels audio “grand public” ne bénéficient pas de matériel dédié. De plus, les applications musicales sont de plus en plus interactives ce qui amène à repenser leurs architectures logicielles. Les travaux effectués lors de ce stage visent à étudier la possibilité d'optimiser dans ces architectures l'utilisation des ressources de calculs, et plus particulièrement celle du CPU, dans un contexte dynamique. Un de nos objectifs était d'obtenir des mesures sur le comportement de ces programmes afin de valider la possibilité d'un traitement unifié des calculs audio périodiques et du contrôle sporadique.

Notre étude s'est volontairement placée dans le cadre d'un système d'aide à la composition, OM7, qui implique à la fois des traitements symboliques lourds et des traitements audio. Notre étude montre que, bien que travaillant dans le contexte d'un langage de haut-niveau (Lisp) qui ne permet de contrôler que très partiellement les ressources du matériel, dans un environnement dynamique (de nombreuses autres tâches s'exécutent dans l'environnement), il est possible d'influer sur l'usage de ces ressources, par exemple en jouant sur le nombre de threads utilisés (et cela bien que ces threads ne soient pas des threads systèmes).

Puisque OM est désormais réactif, il veut proposer une nouvelle vision de l'informatique musicale permettant de nouvelles interactions avec la musique. En effet, des processus compositionnels peuvent interagir avec la séquence d'actions prévues et la modifient dynamiquement. Ce changement se traduit par des opérations de ré-ordonnancement du plan.

Du fait de ce nouveau caractère, OM apporte au contexte temps réel de la lecture, une composante calculatoire du domaine du temps différé où la partition (timeline) peut changer en cours d'exécution. La partition doit toujours s'attribuer un rôle de chef d'orchestre en dictant le comportement que doit adopter le rendu audio. Cela entraine des stratégies d'ordonnancement temps réel qui se veulent nouvelles et différentes.

L'appareillage du code que nous avons développé permet de lancer des campagnes de mesures systématiques. C'est une première étape dans l'étude du comportement de l'ordonnanceur OM7, première étape qui ouvre la voie à son optimisation. Un des objectifs visé à long terme est l'adaptation automatique de l'ordonnanceur aux conditions environnementales dynamiques, ainsi que la mise en place de mises en garde automatiques permettant d'indiquer à l'utilisateur que les traitements prévus sont susceptibles de ne pas respecter les contraintes en temps réel.

À travers cette étude, nous avons pu observer les comportements “cachés” au sein des ordinateurs “grand public”. Dans le cadre du développement d'OM7,

ces comportements “cachés” influent sur le rendu des partitions interactives générées. Nos observations quant à ces comportements peuvent permettre de les contrer dans la mesure du possible.

Cet outil et les résultats proposés pourront être utilisés pour de futurs travaux sur la dégradation adaptative des buffers audio (dans Antescofo) et s'harmonise dans la mesure du possible avec les travaux en cours sur l'adaptabilité des plans d'exécutions de programmes dans OM et la CAO. Une amélioration importante encore possible de l'outil serait de prendre en compte l'ensemble des travaux sur les nouvelles approches d'analyse probabiliste des pires cas de durée d'exécution (probabilistic timing analysis approach) [12].

Bibliographie

[1] Andrea Agostini and Daniele Ghisi. Real-time computer-aided composition with bach. *Contemporary Music Review*, 32(1) :41–48, 2013.

[2] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer-assisted composition at ircam : from patchwork to openmusic. *Computer Music Journal*, 23(3) :59–72, 1999.

[3] Kenneth R Baker and Dan Trietsch. *Principles of sequencing and scheduling*. John Wiley & Sons, 2013.

[4] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5) :720–748, 1999.

[5] Dimitri Bouche and Jean Bresson. Articulation dynamique de structures temporelles pour l'informatique musicale. In *Modélisation des Systèmes Réactifs (MSR 2015)*, 2015.

[6] Dimitri Bouche and Jean Bresson. Planning and scheduling actions in a computer-aided music composition system. In *Scheduling and Planning Applications Workshop (SPARK)*, pages 1–6, 2015.

[7] Dimitri Bouche, Jérôme Nika, Alex Chechile, and Jean Bresson. Computer-aided composition of musical processes. 2015.

[8] Jean Bresson. Reactive visual programs for computer-aided music composition. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages pp–141, 2014.

[9] Jean Bresson, Dimitri Bouche, Jérémie Garcia, Thibaut Carpentier, Florent Jacquemard, John Maccallum, and Diemo Schwarz. Projet efficace : Développements et perspectives en composition assistée par ordinateur. In *Journées d'Informatique Musicale*, 2015.

[10] Jean Bresson and Jean-Louis Giavitto. A reactive extension of the openmusic visual programming language. *Journal of Visual Languages & Computing*, 25(4) :363–375, 2014.

[11] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[12] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis : Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s) :94, 2013.

[13] 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(4) :343–383, 2013.

[14] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

[15] Mikael Laurson and Mika Kuuskankare. Pwgl : a novel visual language based on common lisp, clos and opengl. In *Proceedings of International Computer Music Conference*, pages 142–145, 2002.

[16] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[17] Raphaël Marczak, Myriam Desainte-Catherine, and Antoine Allombert. Real-time temporal control of musical processes. In *Proc. of the International Conferences on Advances in Multimedia (MMEDIA)*, volume 11, page 39, 2011.

[18] James McCartney. Rethinking the computer music language : Supercollider. *Computer Music Journal*, 26(4) :61–68, 2002.

[19] Robert A Moog. Midi : musical instrument digital interface. *Journal of the Audio Engineering Society*, 34(5) :394–404, 1986.

[20] Laurence Nigay and Joëlle Coutaz. Espaces conceptuels pour l'interaction multimédia et multimodale. *Technique et science informatiques*, 15(9) :1195–1225, 1996.

[21] Jérôme Nika, Dimitri Bouche, Jean Bresson, Marc Chemillier, and Gérard Assayag. Guided improvisation as dynamic calls to an offline model. In *Sound and Music Computing (SMC)*, 2015.

[22] Jérôme Nika and Marc Chemillier. Improvisation musicale homme-machine guidée par un scénario temporel. *Technique et Science Informatiques*, 33(7-8) :651–684, 2014.

[23] Project MAC (Massachusetts Institute of Technology). Engineering Robotics Group and ML Dertouzos. *Control robotics : The procedural control of physical processes*. 1973.

[24] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4) :31–43, 2002.

[25] Miller Puckette et al. Pure data : another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.

[26] Xavier Rodet, Pierre Cointe, and Esther Starkier. Formes : composition et ordonnancement de processus. IRCAM, 1985.

[27] S Rooney, D Bauer, and L Garcés-Erice. Building a practical event-scheduler for a multi-processor architecture. In *Proceedings of High Performance Computing and Simulation Conference (HPCS)*, pages 311–318. Citeseer, 2008.

[28] Sean Rooney. Scheduling intense applications most ?surprising ?first. *Science of Computer Programming*, 97 :309–319, 2015.

[29] O Sandred, Mikael Laurson, and Mika Kuuskankare. Revisiting the illiac suite–a rule-based approach to stochastic processes. *Sonic Ideas/Ideas Sonicas*, 2 :42–46, 2009.

[30] Norbert Schnell, Victor Saiz, Karim Barkati, and Samuel Goldszmidt. Of time engines and masters an api for scheduling and synchronizing the generation and playback of event sequences and media streams for the web audio api. In *WAC*, 2015.

[31] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C Buttazzo. *Deadline scheduling for real-time systems : EDF and related algorithms*, volume 460. Springer Science & Business Media, 2012.

[32] Heinrich Taube. Common music : A music composition language in common lisp and clos. *Computer Music Journal*, 15(2) :21–32, 1991.

[33] Daniel Trueman, Perry Cook, Scott Smallwood, and Ge Wang. Plork : the princeton laptop orchestra, year 1. In *Proceedings of the international computer music conference*, pages 443–450, 2006.

[34] Barry Vercoe and Dan Ellis. Real-time csound : Software synthesis with sensing and control. In *Proceedings of the International Computer Music Conference*, pages 209–211, 1990.

[35] ECJE Vidal and Alexander Nareyek. A real-time concurrent planning and execution framework for automated story planning for games. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (September 2011)*, 2011.

[36] Ge Wang. A history of programming and music. *The Cambridge Companion to Electronic Music*, pages 55–71, 2007.

[37] Ge Wang, Perry R Cook, and Spencer Salazar. Chuck : A strongly timed computer music language. *Computer Music Journal*, 2016.

[38] Matthew Wright, Adrian Freed, et al. Open sound control : A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, volume 2013, page 10, 1997.

Annexes

Code de la fonction (macro) permettant de faire une mesure.

```
;t1 correspond au temps de commencement de la tâche.
;threadId récupère l'id du processus utilisant la macro (un type de fonction qui peut transformer le code Lisp lors de la phase de macroexpansion du code).
;tsup représente le temps supposé.
;t2 le temps de fin de tâche.
;idle-core-percent le percentage d'oisiveté.
;time-begin et time-end entant que timestamps.
;freq processeur est mis à jour ultérieurement, le champs de la structure de données prévu pour la fréquence est donc mis a zéro pour le moment.

(defmacro mesure(obj clef thread &body body)
  `(let (res)
     (let ((t1 (round (MACH::mach_absolute_time) 1000))
           (threadId (parse-integer(subseq(reverse(mp:process-name mp:*current-process*)) 0 1)))
           (tsup (+ *begin* (* *cpt-elmt-bpf* (* 1000 *bpf-period* ))))
           (t2)
           (idle-core-percent (- 100(/ (parse-float(multiple-value-bind (out)
                                                     (sys:run-shell-command "ps -A -o %cpu | awk '{s+=$1} END {print s}'"
                                                                            :wait nil
                                                                            :output :stream
                                                                            )
                                                   (with-open-stream (out out)
                                                     (values (read-line out)))))) *logical-core-nb*)))
           (freq-processeur 0)
           (time-begin (date-with-ms)) (time-end 0))
       (setf *cpt-elmt-bpf* (+ *cpt-elmt-bpf* 1))
       (progn
         (setq res (progn ,@body))
         (setq t2 (round (MACH::mach_absolute_time) 1000))
         (setq time-end (date-with-ms))

         (add-list-to-list-of-key  ,obj ,clef (list t1 t2  threadId tsup freq-processeur   (thread-actif-cpt ,obj) (cpt ,obj)  time-begin time-end idle-core-percent)))
       (setf  (cpt ,obj) (+ (cpt ,obj) 1))
       )
     res))
```

Code de la fonction appelée par un thread du pool de threads

```
;self est une instanciation de notre objet thread-pool est la structure thread-pool.
;task est une tâche récupérée depuis une mailbox.
;thread-actif-cpt est un compteur de thread actif.
(defmethod thread-function ((self thread-pool))
  (mp:ensure-process-mailbox) ;creation d'une mailbox ,pour que le thread puisse contenir des tâches dans un message, aux processus si inexistante.
  (loop
   (let ((occthread)
         (task (mp:mailbox-read (taskqueue self))))
     (if (not task)
         (mp:process-wait-local "Waiting for request asleep" 'mp:mailbox-not-empty-p (taskqueue self))
       (progn
         (incf (thread-actif-cpt (monitor-time self)))
         (mesure-all-time-us (monitor-time self) (cpt (monitor-time *engine*)) mp:*current-process*
                             (funcall task))
         (decf (thread-actif-cpt (monitor-time self))))
       ))))
```