Rapport de stage

Amélioration des stratégies d'ordonnancement dans le compilateur FAUST

Auteur :

Simon Jacquin simon.jacquin@atiam.fr

Encadrants :

Yann Orlarey orlarey@grame.fr
Tanguy Risset tanguy.risset@insa-lyon.fr

Dates du stage :

3 mars – 5 septembre 2025

Sujet du stage

Le stage se déroule au sein de l'équipe Emeraude (INRIA/INSA/GRAME) à Lyon, qui travaille notamment sur le développement du compilateur Faust (`https://faust.grame.fr/`), un langage de programmation dédié au traitement et à la synthèse du signal audio.

Dans une phase de la compilation, les calculs d'un programme Faust sont représentés sous forme de *DAG* (Directed Acyclic Graph) d'instructions, une structure qui décrit les dépendances entre ces différentes instructions. Ce DAG est ensuite transformé, via un *tri topologique*, en une séquence d'instructions exécutables.

L'ordre topologique choisi a un impact direct sur les performances du programme compilé, particulièrement en raison des caractéristiques des processeurs modernes, comme l'utilisation des caches. Le compilateur Faust intègre déjà plusieurs stratégies pour optimiser cet ordonnancement, par exemple en minimisant le nombre d'instructions séparant la production d'un résultat de ses utilisations. Cependant, aucune stratégie ne se révèle systématiquement optimale pour un ensemble représentatif de programmes Faust.

Objectif du stage

Le stage vise à concevoir et à expérimenter de nouvelles stratégies d'ordonnancement dans le compilateur Faust. L'étudiant(e) devra :

- Analyser les stratégies existantes et proposer des approches alternatives en précisant les critères sur lesquels elles se basent.
- Implémenter ces nouvelles stratégies dans le compilateur Faust.
- Développer une infrastructure de test pour évaluer de manière systématique l'impact des différentes stratégies sur des architectures CPU variées.
- Comparer les performances des stratégies sur un ensemble représentatif de programmes Faust et identifier les cas d'usage ou domaines d'application pour lesquels chaque stratégie est la plus efficace.

Table des matières

1 Introduction **3**
1.1 Le langage Faust . . . 3
1.2 Graphe de calcul et tri topologique . . . 4

2 État de l'art **7**
2.1 Stratégies déjà implantées . . . 7
2.2 Mesure du temps de calcul . . . 7
2.3 Recherche d'explications . . . 8
2.3.1 Influence de l'utilisation du cache . . . 10
2.3.2 Influence de la taille de l'assembleur produit . . . 10
2.3.3 Influence du choix du compilateur C++ . . . 10
2.4 Conclusion . . . 11

3 Protocole de mesure **12**
3.1 Quelques rappels utiles sur l'architecture des CPU . . . 12
3.1.1 Pipelining . . . 12
3.1.2 Instruction-level parallelism (ILP) . . . 13
3.1.3 Data-level parallelism : SIMD . . . 13
3.1.4 Types de mémoire . . . 14
3.2 Performance Monitoring Units . . . 14
3.3 Faust Scheduling Test Suite . . . 15
3.3.1 Modes d'exécution . . . 15
3.3.2 Graphiques de performances . . . 16
3.3.3 Tests de validité . . . 16
3.4 Observations . . . 16
3.4.1 Quantité d'instructions et pipeline stalls . . . 17
3.4.2 Bilan d'utilisation des ressources . . . 20
3.5 Conclusion . . . 24

4 Améliorations **25**
4.1 Le tri topologique dans la génération de code assembleur . . . 25
4.2 Un List Scheduling pour Faust . . . 26
4.3 Mode vectoriel et séparation des boucles . . . 27

5 Conclusion et perspectives **32**

Chapitre 1

Introduction

1.1 Le langage Faust

Le langage Faust [1] est un langage de programmation fonctionnel destiné à la synthèse et au traitement du son numérique. Il est particulièrement adapté à la conception de synthétiseurs, instruments de musique, et effets sonores.

Un programme Faust décrit une fonction mathématique représentant un processus sur des signaux (souvent abrégé DSP, pour *Digital Signal Processor*), qui peut être compilé vers un programme impératif équivalent dans un large choix de langages (C, C++, LLVM IR, WebAssembly, Rust...). Ces signaux sont décrits à l'échantillon près, ce qui permet de représenter des fonction de traitement de signaux de bas niveau comme des filtres récursifs, là où des langages de plus haut niveau comme SuperCollider ou Max/MSP implémentent ces filtres au moyen de primitives.

La documentation du langage (en anglais) est consultable à cette adresse : `https://faustdoc.grame.fr/`. À titre d'exemple, voici comment on peut synthétiser un son de corde pincée avec l'algorithme de Karplus-Strong en Faust :

```
import("stdfaust.lib");

process = button("play") : resonator(80, 1)
with {
    resonator(d, a) = (+ : @(d-1)) ~ (average : *(a));
    average(x) = x+x' : *(0.5);
};
```

FIGURE 1.1 – Implémentation en Faust de l'algorithme de Karplus-Strong

La compilation du code Faust vers un langage impératif peut se décomposer en six étapes principales [1] :

1. Analyse du code source : on construit un arbre syntaxique à partir du code source.
2. Évaluation des block-diagrams : le processus principal est évalué pour être traduit en un schéma fonctionnel (*block-diagram*) aplati qui ne contient que des blocs élémentaires (fonctions mathématiques de bases, entrées/sorties, signaux constants...). Ces block-diagrams peuvent être visualisés pour représenter un programme Faust : la figure 1.2 montre le diagramme correspondant au programme ci-dessus.
3. Les block-diagrams sont transformés en signaux : on détermine l'équation mathématique de chaque signal de sortie en fonction des signaux d'entrée ;

1. `https://faust.grame.fr/`

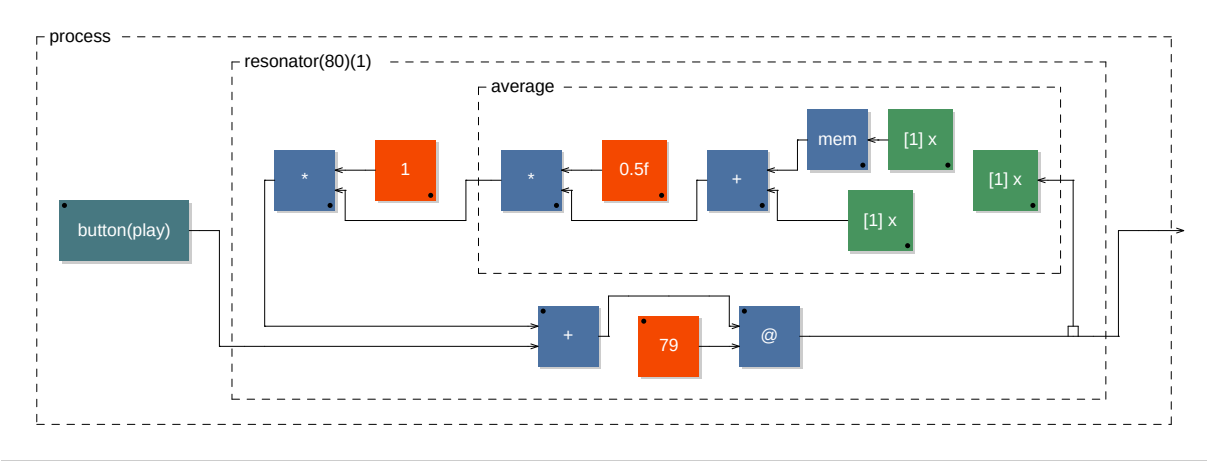

FIGURE 1.2 – Block-diagram du programme de la figure 1.1.

4. Assignation des types : on assigne des types aux signaux, selon plusieurs catégories :
   — la nature du signal (entier ou flottant) ;
   — les valeurs minimale et maximale qu'il peut prendre ;
   — le moment de calcul du signal : à la compilation, à l'initialisation, ou à l'exécution ;
   — sa vitesse : un signal peut être constant (calculé une fois), lent (calculé à chaque bloc d'échantillons), ou rapide (calculé pour chaque échantillon) ;
   — son parallélisme : les échantillons d'un signal peuvent être calculés en parallèle si le signal n'a pas de dépendances récursives.
5. Analyse d'occurrences : on analyse dans quel contexte chaque sous-expression des signaux de sortie est utilisée pour découvrir des expressions communes. Si une sous-expression complexe est découverte, on génère une variable de cache qui permettra de réutiliser le résultat intermédiaire.
6. Génération de code : on traduit les expressions annotées en une séquence d'instructions impératives qui permettent de calculer les signaux de sortie en fonction des signaux d'entrée.

1.2 Graphe de calcul et tri topologique

Depuis 2020, le compilateur utilise un graphe orienté pour représenter les calculs au début de la dernière étape [2]. Dans ce graphe, chaque sommet représente une opération élémentaire (addition, retard, entrée/sortie...), et les arrêtes relient les opérations à leurs opérandes. On peut visualiser ce graphe grâce à l'option `-sg` : la figure 1.3 montre le graphe produit en compilant le programme de la figure 1.1.

Les arrêtes du graphe comportent une indication de temps qui représente le décalage, en échantillons, entre le calcul et son utilisation. En ôtant du graphe toutes les dépendances dont le temps est non nul, on obtient le graphe des dépendances instantanées qui est un graphe orienté acyclique (ou DAG en anglais, pour *Directed Acyclic Graph*), représenté figure 1.4. On peut alors calculer les signaux de sortie en exécutant, pour chaque échantillon, l'ensemble des instructions de ce nouveau graphe dans un ordre qui respecte les relations de dépendances.

Cet ordre s'appelle un ordre topologique. Formellement, on génère une séquence S d'instructions telle que, si le graphe contient une arrête qui va d'une instruction i à une instruction j, alors i apparait après j dans S.

La figure 1.5 montre le code généré par deux ordres topologiques différents pour le programme de la figure 1.1. Les parcours en profondeur et en largeur sont détaillés dans le chapitre 2.

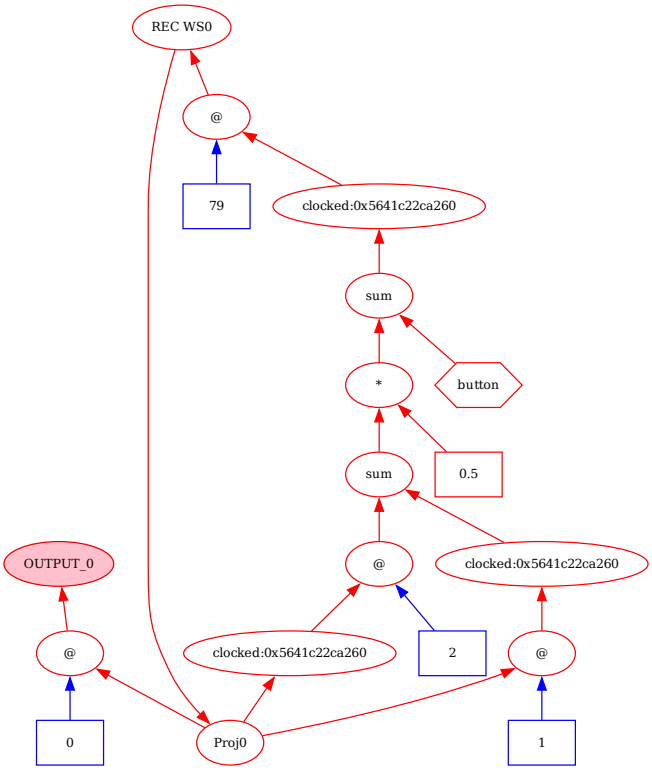

FIGURE 1.3 – Graphe orienté du programme de la figure 1.1. Les parties en bleu représentent des calculs sur des entiers, les parties en rouge des calculs sur des flottants. Le graphe fait apparaître des cycles.

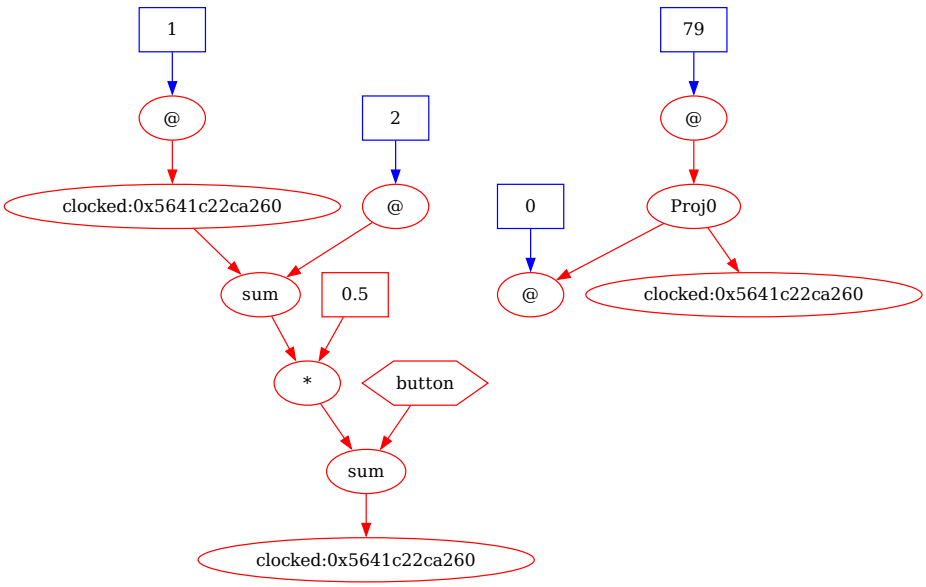

FIGURE 1.4 – Une fois les dépendances non instantanées supprimées, le graphe de la figure 1.3 devient un graphe orienté acyclique.

Parcours en profondeur

```
void compute (int count, float** input, float** output) {
    float fRec0SE[2];
    float fSlow0BE = float(fbutton0); // Zone 2, step: 6
    float fTemp0SE; // step: 11
    float fVec3SE[3];
    float fVec2SE[2];
    float* output0 = output[0]; // Zone 3
    fRec0SE[1] = fRec0SEState;
    fVec3SE[1] = fVec3SEState[0];
    fVec3SE[2] = fVec3SEState[1];
    fVec2SE[1] = fVec2SEState;
    for (int i=0; i<count; i++) {
        fRec0SE[0] = fVec0SE[(IOTA0-79)&127];
        fTemp0SE = (fSlow0BE + -fVec2SE[1]); // step: 11
        fVec0SE[IOTA0&127] = (
        (0.5f * (fRec0SE[1] + fVec3SE[2])) +
        (fTemp0SE * float((fTemp0SE > 0.0f)))
        );
        fVec3SE[0] = fRec0SE[0];
        fVec2SE[0] = fSlow0BE;
        output0[i] = (FAUSTFLOAT)(fRec0SE[0]);
        // post processing
        fVec2SE[1] = fVec2SE[0];
        fVec3SE[2] = fVec3SE[1];
        fVec3SE[1] = fVec3SE[0];
        fRec0SE[1] = fRec0SE[0];
        ++IOTA0;
    }
    fRec0SEState = fRec0SE[1];
    fVec3SEState[0] = fVec3SE[1];
    fVec3SEState[1] = fVec3SE[2];
    fVec2SEState = fVec2SE[1];
}
```

Parcours en largeur

```
void compute (int count, float** input, float** output) {
    float fSlow0BE = float(fbutton0); // Zone 2, step: 8
    float fVec0BE[2];
    float fVec3SE[2];
    float fTemp0SE; // step: 17
    float fVec2SE[3];
    float* output0 = output[0]; // Zone 3
    fVec0BE[1] = fVec0BEState;
    fVec3SE[1] = fVec3SEState;
    fVec2SE[1] = fVec2SEState[0];
    fVec2SE[2] = fVec2SEState[1];
    for (int i=0; i<count; i++) {
        fVec0BE[0] = fSlow0BE;
        fVec3SE[0] = fVec4SE[(IOTA0-79)&127];
        fTemp0SE = (fSlow0BE + -fVec0BE[1]); // step: 17
        fVec2SE[0] = fVec3SE[0];
        fVec4SE[IOTA0&127] = (
                (0.5f * (fVec3SE[1] + fVec2SE[2])) +
                (fTemp0SE * float((fTemp0SE > 0.0f)))
        );
        output0[i] = (FAUSTFLOAT)(fVec3SE[0]);
        // post processing
        fVec2SE[2] = fVec2SE[1];
        fVec2SE[1] = fVec2SE[0];
        fVec3SE[1] = fVec3SE[0];
        ++IOTA0;
        fVec0BE[1] = fVec0BE[0];
    }
    fVec0BEState = fVec0BE[1];
    fVec3SEState = fVec3SE[1];
    fVec2SEState[0] = fVec2SE[1];
    fVec2SEState[1] = fVec2SE[2];
}
```

FIGURE 1.5 – Ces deux implémentations du programme karplus comportent les mêmes instructions, mais correspondent à des ordres topologiques différents.

Plusieurs stratégies de tri topologique ont été testées, et on a immédiatement observé que le code produit, une fois compilé, ne s'exécutait pas avec la même rapidité selon l'ordre des instructions. Cependant, en moyenne, les stratégies testées obtiennent de meilleurs résultats que les versions précédentes qui n'utilisaient pas cette représentation en DAG [2]. On mesure malgré tout une grande variabilité entre les stratégies étudiées, et aucune ne semble systématiquement plus performante que les autres.

Le chapitre 2 reprend ces observations. On y explique les différents tris topologiques déjà implémentés, puis on montre une première série de mesures qui a orienté notre compréhension du problème. Dans le chapitre 3, on détaille un protocole de mesures élaboré spécifiquement pour ce problème et les observations qu'il a rendues possibles. On montre que ces observations permettent d'expliquer les différences mesurées. Enfin, le chapitre 4 explore des pistes d'amélioration à la lecture de ces résultats.

Chapitre 2

État de l'art

2.1 Stratégies déjà implantées

Dans une version expérimentale [1], le compilateur Faust propose quatre méthodes différentes de tri topologique (appelées *scheduling strategies*), accessibles avec l'option `-ss` suivie d'un nombre entre 0 et 3 :

0. Un parcours en profondeur (*deep-first*) : lorsqu'un calcul a plusieurs dépendances, chaque dépendance est évaluée entièrement avant d'évaluer la suivante.
1. Un parcours en largeur (*breadth-first*) : on décompose le graphe en étages. Le premier étage contient les calculs finaux, le second étage contient leurs dépendances directes, le troisième contient les dépendances de ces dépendances, et ainsi de suite jusqu'à arriver aux calculs initiaux. On planifie ensuite chaque étage en partant du dernier et en finissant par le premier.
2. Un parcours entrelacé (*interleaved*), appelé *special* dans le code source : comme le parcours en profondeur, lorsqu'un calcul a plusieurs dépendances, il construit la séquence d'instructions complète de chaque dépendance, mais les instructions des séquences ainsi créées sont ensuite entrelacées entre elles.
3. Un parcours en largeur inversé (*reverse-breadth-first*) : comme pour un parcours en largeur, on décompose le graphe en étages, mais en partant des calculs initiaux. Le premier étage contient tous les calculs n'ayant aucune dépendance, le second contient ceux ayant uniquement des dépendances dans le premier étage, et ainsi de suite jusqu'à arriver aux calculs finaux. On planifie alors chaque étage en partant du premier et en finissant par le dernier.

2.2 Mesure du temps de calcul

Pour dresser un état des lieux de ces différentes stratégies, nous avons tout d'abord utilisé un outil de mesure du temps de calcul (*benchmark*) déjà disponible pour Faust : `fcbenchtool` [2]. Ce programme produit un exécutable qui recherche le temps minimum d'exécution de la boucle principale du DSP compilé, sur un buffer de 44100 échantillons. Un temps est considéré comme le temps minimal s'il n'a pas été battu après 1000 itérations successives.

Nous avons compilé les 246 exemples fournis avec le projet Faust avec chacune des quatre stratégies disponibles, puis mesuré le temps d'exécution minimal de chaque stratégie, produisant ainsi un total de 984 mesures de performances.

1. `https://github.com/grame-cncm/faust/tree/master-dev-ocpp-od-fir-2-FIR5`
2. `https://github.com/orlarey/faustcompilerbenchtool`

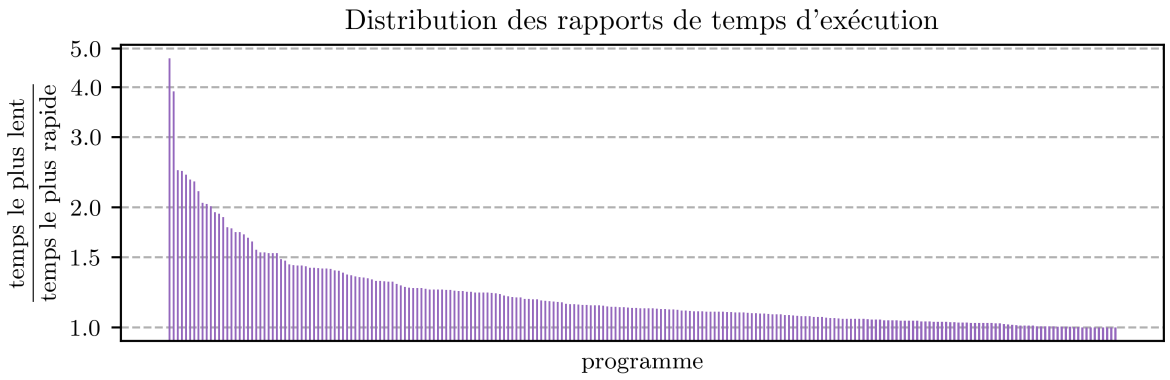

FIGURE 2.1 – Distribution du rapport de temps d'exécution entre la stratégie la plus efficace et la stratégie la moins efficace pour les 246 exemples testés. Lorsque ce rapport vaut 1, toutes les stratégies s'exécutent à la même vitesse. Lorsqu'il vaut 2, cela signifie qu'une des stratégies s'est exécutée deux fois plus vite qu'une autre dans nos mesures.

Le code Faust est compilé avec l'option `-lang ocpp`[3] ; le code C++ est ensuite compilé avec `clang++` (version 20) et les options `-O3`, `-ffast-math` et `-march=native`.

Les tests ont été effectués sur un ordinateur personnel fonctionnant avec un noyau Linux (version 6.13.8), équipé d'un processeur Intel Core i5-8265U de 2018.

Une première mesure qui nous a semblé intéressante est celle du rapport de temps d'exécution entre la stratégie la plus lente et la stratégie la plus rapide. La figure 2.1 montre la distribution de ce rapport pour les exemples testés. On remarque que l'écart est souvent significatif : 169 programmes sur 246 s'exécutent avec un écart supérieur à 5%, 62 avec un écart supérieur à 25%, et on observe même 11 cas critiques où l'écart dépasse 100%, pour culminer à 372% avec le programme `vocoder.dsp`.

La figure 2.2 montre, dans le détail, le temps de calcul de chaque stratégie par rapport au temps moyen pour les 27 programmes où l'écart dépasse 50%. À première vue, aucune stratégie ne semble se démarquer par sa rapidité ni par sa lenteur. Ceci se vérifie en mesurant, pour les 246 programmes, la performance moyenne de chaque stratégie par rapport aux autres (tableau 2.3) : les écarts à la moyenne sont inférieurs à 2%, ils ne sont donc pas significatifs compte tenu du nombre de mesures effectuées. On peut tout de même noter que la stratégie d'entrelacement semble plus instable, avec un écart type deux fois plus élevé que les trois autres.

2.3 Recherche d'explications

La version du compilateur Faust utilisée pour produire ces tests affiche, à chaque compilation, une estimation chiffrée de la performance de l'ordonnancement produit. Cette estimation repose sur l'idée suivante : plus un calcul est placé loin de ses dépendances, plus il y a de chances pour que le résultat de ce calcul ne se trouve plus dans les registres, voire dans le cache, au moment où il sera utilisé. On calcule donc la somme du carré des distances entre chaque relation de dépendance pour pénaliser l'éloignement entre un calcul et son utilisation.

Pour vérifier la pertinence de cet indicateur, on a calculé le coefficient de corrélation entre, d'une part, le temps de calcul de chaque stratégie rapporté au temps de calcul moyen des quatre stratégies pour le même programme, et d'autre part l'estimation du coût de chaque stratégie normalisée de la même manière. Sur l'ensemble des 246 programmes, ce coefficient vaut -0.0502 : cet indicateur ne semble finalement donner aucune information sur le temps d'exécution réel.

3. La génération de code C++ historique de Faust, par opposition à la génération plus récente basée sur la *Faust Intermediate Representation* (FIR)

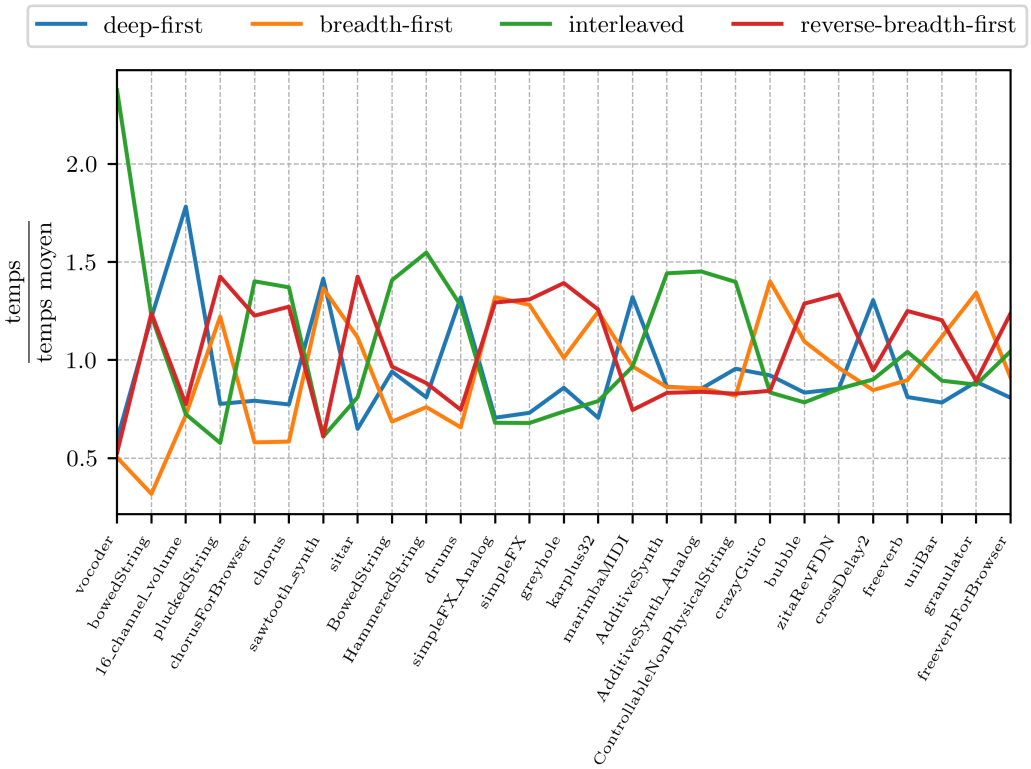

FIGURE 2.2 – Rapport entre le temps de calcul produit par une stratégie et le temps de calcul moyen de toutes les stratégies, sur l'ensemble des programmes pour lesquels on a mesuré un rapport de 1.5 ou plus entre la stratégie la plus lente et la stratégie la plus rapide (voir figure 2.1).

Stratégie	Temps relatif moyen	Écart type
Deep-first	1.01	0.13
Breadth-first	0.99	0.13
Interleaved	0.98	0.24
Reverse breadth-first	1.02	0.13

FIGURE 2.3 – Temps relatif moyen et écart type du temps d'exécution de chaque stratégie par rapport au temps d'exécution moyen des quatre stratégies. Le temps relatif moyen correspond à la moyenne, sur les 246 programmes testés, du temps d'exécution d'une stratégie divisé par la moyenne des temps d'exécution de toutes les stratégies pour le même programme. Les quatre stratégies ne semblent pas se démarquer par cette mesure, cependant la stratégie *Interleaved* produit des performances plus variables

2.3.1 Influence de l'utilisation du cache

Afin de vérifier l'intuition selon laquelle une meilleure utilisation du cache du processeur serait à l'origine des différences observées, nous avons utilisé l'outil `perf` [4], fourni avec le noyau Linux, qui permet de compter certains types d'évènements se produisant lors de l'exécution d'un processus, et notamment les défauts de cache (*cache-miss*), qui se produisent lorsque le processus essaye d'accéder à une donnée qui n'est pas présente dans le cache. Cette notion d'évènement et la technique utilisée pour les mesurer sont développées au chapitre suivant, dans la section 3.2.

L'évènement le mieux corrélé avec les différences de temps d'exécution, en appliquant le même procédé qu'à la section que précédemment (où l'on rapporte les mesures de chaque programme à la moyenne des mesures des autres stratégies pour ce même programme), semble effectivement être un évènement nommé `L1-dcache-load-miss`, dont le facteur de corrélation vaut 0.4 sur l'ensemble des programmes considéré, et monte à 0.76 en ne considérant que les 27 programmes de la figure 2.2. Cet évènement se produit lorsque le processeur a besoin de lire une variable qui ne se trouve pas dans le cache L1, le cache de données de plus bas niveau. Les différents niveaux de cache seront également détaillés au chapitre suivant, section 3.1.4.

2.3.2 Influence de la taille de l'assembleur produit

Ces résultats doivent cependant être considérés avec prudence. En effet, si la quantité de défauts de cache change d'un programme à l'autre, cela peut tout simplement être dû à un nombre d'accès mémoire globalement plus élevé. En effet, nous avons mesuré que la quantité totale d'instructions exécutées lors d'un benchmark pouvait varier fortement entre les quatre différentes stratégies d'ordonnancement.

En mesurant la taille du code assembleur produit pour le programme `vocoder` par les différentes stratégies, on a observé des différences très significatives : d'une boucle principale de 859 instructions pour le parcours en largeur, elle passe à 4179 instructions pour le parcours entrelacé. Or le rapport de performances entre ces deux stratégies est du même ordre de grandeur.

Pour suivre cette nouvelle piste, nous avons donc mesuré le nombre de lignes de code assembleur de la boucle principale d'exécution pour chaque stratégie et pour chaque programme des exemples. Puis nous avons calculé, selon le principe appliqué précédemment, la corrélation entre nombre d'instructions et temps d'exécution. Nous obtenons cette fois ci un coefficient de 0.4. Il semblerait donc que l'ordre des instructions ait une influence sur la capacité du compilateur C++ à générer un code optimal.

L'éloignement entre un calcul et son utilisation reste une explication possible pour cette différence de taille en raison des *spills* de registres : lorsqu'une variable ne peut être utilisée immédiatement après son calcul, le compilateur génère des instructions supplémentaires pour la sauvegarder en mémoire. Ce processus sera étudié plus en détail dans le chapitre 3.

2.3.3 Influence du choix du compilateur C++

Le compilateur jouant manifestement un rôle prépondérant ici, nous avons relancé les mêmes mesures qu'à la section 2.2 en utilisant cette fois le compilateur GCC version 15 (figure 2.4). Les différences de temps de calcul observées sont significatives : en appliquant toujours la même fonction de corrélation entre les temps de calculs mesurés avec ces deux compilateurs, on n'obtient qu'un coefficient de 0.3 pour l'ensemble des programmes d'exemple. Un protocole de mesure robuste devra donc pouvoir comparer les performances du code généré avec plusieurs compilateurs.

4. `https://perfwiki.github.io/main/`

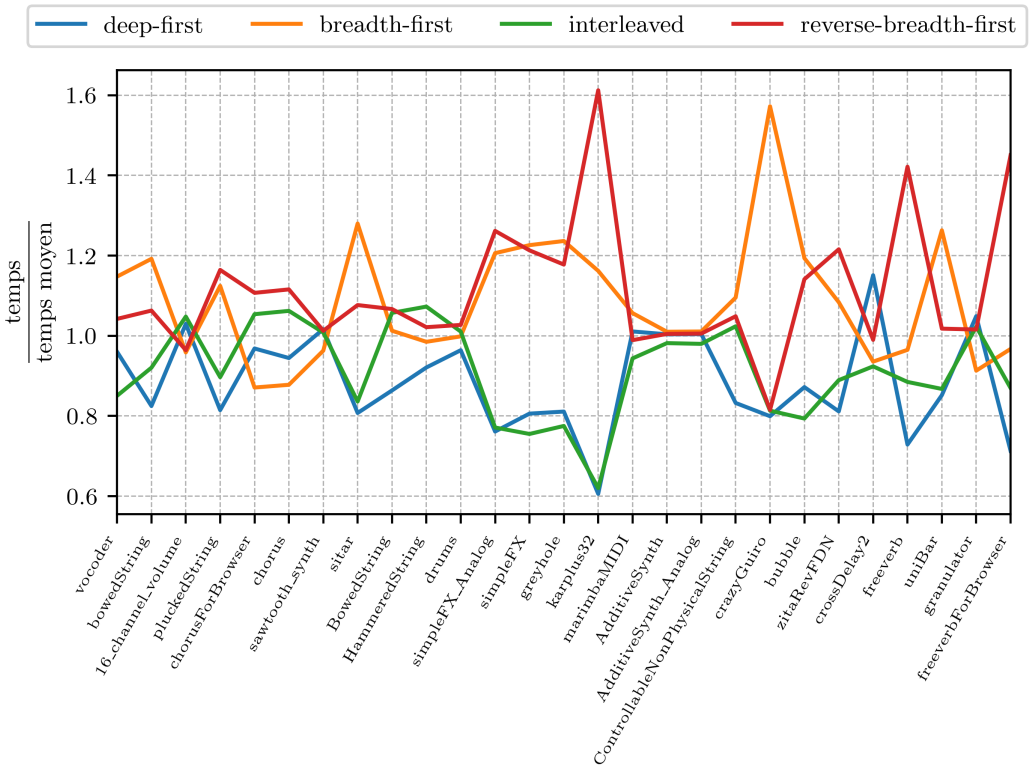

FIGURE 2.4 – Rapport entre le temps de calcul produit par une stratégie et le temps de calcul moyen de toutes les stratégies, en utilisant le compilateur GCC version 15, pour les programmes de la figure 2.2.

2.4 Conclusion

Ces mesures préliminaires nous ont permis de reproduire les observations à l'origine de ce sujet de stage. Elles suggèrent également que les causes des différences de temps de calcul mesurées sont plus diverses que ce qu'on pouvait penser initialement : si l'utilisation des registres et du cache ont clairement un impact, d'autres paramètres entrent également en jeu, et notamment la capacité des compilateurs à optimiser le code machine généré.

Dans le chapitre suivant, on construit une méthodologie permettant de mesurer de manière systématique l'impact des différentes stratégies d'ordonnancement sur l'usage des ressources d'un processeur, ce qui nous permettra d'isoler les mécanismes à l'œuvre.

Chapitre 3

Protocole de mesure

3.1 Quelques rappels utiles sur l'architecture des CPU

Pour étudier les mécanismes qui peuvent amener le même code à s'exécuter à des vitesses différentes selon l'ordre dans lequel il est spécifié, il est nécessaire de connaître certains mécanismes de base présents dans la grande majorité des processeurs aujourd'hui. On va s'intéresser ici aux principales techniques que les processeurs modernes implémentent pour augmenter le débit des instructions : pipelining, exécution out-of-order, superscaling, et vectorisation. On rappellera ensuite l'organisation de la mémoire qui implique différents niveaux de cache, dont la latence mais aussi la taille diminue à mesure qu'on se rapproche d'un cœur du CPU.

3.1.1 Pipelining

Le pipelining est la technique fondamentale permettant aux CPU d'atteindre de grandes vitesses d'exécution : le traitement de chaque instruction est décomposé en étapes élémentaires, pouvant être exécutées simultanément pour des instructions successives [3]. Une décomposition classique se fait en cinq étapes : *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execution* (EX), *Memory access* (MEM), *Write-back* (WB).

Lorsque la première étape (IF) d'une instruction n est terminée, pendant qu'on passe à la deuxième étape, on peut simultanément exécuter la première étape de l'instruction $n+1$. La figure 3.1 représente un cas idéal où le CPU peut ainsi exécuter en même temps les 5 étapes de 5 instructions différentes.

Cycle	1	2	3	4	5	6	7	8	9
Instruction n	IF	ID	EXE	MEM	WB				
Instruction $n+1$		IF	ID	EXE	MEM	WB			
Instruction $n+2$			IF	ID	EXE	MEM	WB		
Instruction $n+3$				IF	ID	EXE	MEM	WB	
Instruction $n+4$					IF	ID	EXE	MEM	WB

FIGURE 3.1 – Pipeline simple en 5 étapes

Bien entendu, il existe de nombreuses situations [4] qui empêchent de réaliser ce cas idéal :

— *Read-after-write* (RAW) : Lorsque l'opération $n+1$ lit le contenu d'un registre dans lequel l'opération n est censée écrire. Dans ce cas le pipeline est bloqué jusqu'à ce que l'opération n soit entièrement terminée. On peut cependant économiser quelques cycles en récupérant cette valeur directement à l'étape EXE.

— *Write-after-read* (WAR) : Lorsque l'opération $n+1$ écrit dans un registre et écrase une valeur que l'instruction n n'a pas encore eu le temps de lire. Ce problème n'est pas véritablement un problème de dépendance et peut être résolu en utilisant un registre différent.

— Lorsqu'une instruction contient une branche, on ne peut pas savoir quelle sera l'instruction suivante à exécuter avant d'avoir entièrement évalué son résultat. Dans ce cas, la plupart des CPU ont recours à l'exécution spéculative : une des deux branches possibles est choisie en fonction d'une heuristique, et les instructions ainsi exécutées ne seront validées que si ce choix s'avère juste après évaluation de la branche.

3.1.2 Instruction-level parallelism (ILP)

Dans les exemples qui précèdent, les instructions sont exécutées dans le même ordre que celui dans lequel elles apparaissent dans le programme. Cependant, dans la mesure où elles ne dépendent pas toutes les unes des autres, la plupart des CPU sont capables de les réordonner pour exploiter au mieux leur pipeline : on parle d'exécution out-of-order (OOO). Ce mécanisme permet de garder le processeur occupé lorsqu'une instruction est dans l'attente d'une de ses opérandes, que ce soit le résultat d'une instruction précédente ou une donnée qui doit être récupérée dans la mémoire.

La plupart des CPU modernes sont superscalaires : ils sont capables d'exécuter entre 2 et 6 instructions simultanément, en multipliant les unités évaluant chaque étape du pipeline. La figure 3.2 montre l'exemple précédent réalisé dans un pipeline superscalaire capable d'exécuter deux instructions à la fois. Combiné à l'exécution out-of-order, ce mécanisme permet de multiplier le débit d'instructions réalisé par un seul cœur de processeur.

Cycle	1	2	3	4	5	6	7
Instruction n	IF	ID	EXE	MEM	WB		
Instruction $n+1$	IF	ID	EXE	MEM	WB		
Instruction $n+2$		IF	ID	EXE	MEM	WB	
Instruction $n+3$		IF	ID	EXE	MEM	WB	
Instruction $n+4$			IF	ID	EXE	MEM	WB

FIGURE 3.2 – Pipeline superscalaire de degré 2

3.1.3 Data-level parallelism : SIMD

Les instructions de type SIMD (single instruction, multiple data) permettent d'appliquer la même opération à plusieurs données indépendantes en parallèle. Elles sont particulièrement adaptées au traitement du signal numérique qui demande souvent d'appliquer la même opération à un flux continu de nombres. Sur x86, il s'agit des instructions SSE (Streaming SIMD Extensions) et AVX (Advanced Vector Extensions). Sur les processeurs modernes, elles permettent de grouper les calculs par vecteurs de 2, 4, 8 ou 16 opérandes en parallèle[3]. On parlera d'instructions vectorisées.

C'est au compilateur C++ de générer ces instructions. Le compilateur CLANG supporte des directives (pragma) permettant d'indiquer qu'une boucle devrait être vectorisée [1]. Ainsi, la boucle suivante pourra être compilée sous la forme d'instructions vectorisées :

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < nsamples; i++) {
    output[i] = input[i] * gain;
}
```

Cependant, la plupart du temps, cette indication n'est pas nécessaire : le compilateur est capable de découvrir tout seul les boucles susceptibles d'être vectorisées. Il faut toutefois noter

1. https://llvm.org/docs/Vectorizers.html

que la quantité d'instructions n'est pas nécessairement un multiple de la taille du vecteur de l'instruction SIMD utilisée : dans l'exemple ci-dessus, si `nsamples` vaut 15 mais que la boucle est implémentée en utilisant une instruction SIMD de taille 4, les trois dernières itérations ne pourront pas utiliser la vectorisation et devront se faire une à une. Le compilateur doit donc générer deux fois le code de la boucle : une version vectorisée et une version scalaire pour exécuter les dernières itérations.

Une autre forme d'autovectorisation est la vectorisation SLP (Superword-Level Parallelism), qui ne nécessite pas de boucle. Elle consiste à trouver plusieurs instructions similaires et indépendantes et à les regrouper en une seule instruction vectorielle. Lorsque Faust est utilisé en mode scalaire (ce qui est le cas ici), on rencontre surtout ce type de vectorisation.

3.1.4 Types de mémoire

La mémoire d'un ordinateur est organisée en différents niveaux. Les niveaux les plus bas sont les plus rapides et les plus petits. Les registres du processeur constituent le niveau de mémoire le plus bas et le plus petit. Viennent ensuite deux ou trois niveaux de cache (appelés L1, L2 et L3 le cas échéant), qui vont de quelques dizaines de kilooctets à quelques mégaoctets. À la fin, on trouve la mémoire principale (ou RAM, pour random-access memory), beaucoup plus lente mais d'une capacité qui monte en général à plusieurs gigaoctets. La figure 3.3 résume ces différents niveaux de mémoire sur une architecture de PC typique, avec leurs latences et leur taille.

Niveau de mémoire	Latence en cycles (temps)	Taille
Cache L1	4 cycles (∼1 ns)	64 Ko
Cache L2	10-25 cycles (5-10 ns)	256 Ko
Cache L3	40 cycles (20 ns)	2-4 Mo
Mémoire principale	> 200 cycles (100 ns)	> 4 Go

FIGURE 3.3 – Latences typiques d'une architecture mémoire [4] [3]

Les niveaux de mémoire sont, normalement, inclusifs [3] : chaque niveau contient un sous-ensemble des données du niveau immédiatement supérieur. Ils sont utilisés par le processeur selon deux propriétés fondamentales :

— **Localité temporelle** : quand un programme accède à une donnée mémoire, il y a de fortes chances qu'il accède à nouveau à cette donnée dans un futur proche. On s'arrange donc pour qu'elle soit présente dans le plus bas niveau possible du cache.

— **Localité spatiale** : quand un programme accède à une donnée dans la mémoire, il aura tendance à accéder ensuite à des données situées dans la même zone mémoire. Cela signifie qu'en plus de charger dans le cache la donnée que le programme a demandée, on va typiquement charger un ou plusieurs blocs mémoire (*cache line*) de données qui lui sont contigües. Une ligne de cache fait typiquement 64 octets.

Lorsqu'un programme cherche à accéder à une donnée qui ne se trouve pas dans le cache de plus bas niveau, on parle de défaut de cache (*cache-miss* en anglais). Une ligne de cache doit donc être chargée à partir d'un niveau supérieur (L2, L3 ou dans la mémoire principale), puis insérée au niveau L1 à la place d'une ligne plus ancienne. Selon le niveau auquel on doit remonter pour récupérer cette donnée, la latence sera d'autant plus longue.

3.2 Performance Monitoring Units

Pour comprendre dans le détail le comportement de nos programmes Faust, on a besoin d'outils pour analyser l'utilisation des ressources du CPU que l'on vient de mentionner. On va chercher à savoir, d'une part, si nos programmes permettent d'utiliser au mieux le pipeline, le

superscaling du processeur et les instructions vectorielles du processeur ; et d'autre part si leur utilisation de la mémoire est suffisamment localisée pour minimiser les défauts de cache.

Tous les CPU modernes fournissent des outils capables de mesurer leur utilisation : ce sont les Performance Monitoring Units (PMU). Les PMU contiennent un ensemble de Performance Monitoring Counters (PMC) qui peuvent être utiliser pour compter les occurrences d'évènements arbitraires [4], tels que :

— Le nombre de cycles écoulés ;
— Le nombre d'instructions exécutées ;
— Le nombre de cycles pendant lesquels une, deux, trois ou quatre instructions sont exécutées en même temps (pour un processeur superscalaire) ;
— Le nombre de cycles passés sans exécuter d'instructions (*pipeline stalls*) ;
— Le nombre d'opérations à virgule flottante (*flops*) exécutées, de manière scalaire ou vectorielle ;
— Le nombre d'accès mémoire aboutissant à un niveau donné du cache (L1, L2 ou L3 sur x86-64) (*cache-hit*) ;
— Le nombre d'accès mémoire n'aboutissant pas à un niveau donné du cache (*cache-miss*) ;
— Et bien d'autres, en fonction des choix des fabricants.

Le noyaux Linux expose ces registres via des appels système, et la bibliothèque libPFM [2], écrite en langage C, offre une interface pour y accéder facilement.

Cependant, chaque processeur implémente son propre jeu d'évènements et bien que la libPFM soit capable de regrouper sous un même nom des évènements similaires d'architectures différentes, il est difficile de concevoir un outil générique comptant exactement les mêmes évènements sur toutes les architectures. De plus, si le noyau Linux donne un accès aux PMU, ce n'est pas le cas du noyau XNU utilisé par macOS. Ces particularités peuvent rendre les mesures compliquées à reproduire d'une machine à l'autre.

3.3 Faust Scheduling Test Suite

Les mesures qui vont suivre ont toutes été effectuées avec un outil développé pour l'occasion, appelé Faust Scheduling Test Suite [3]. Écrit en Python, il génère, à partir d'un fichier Faust, un ensemble de sources C++ correspondant chacune à un ordonnancement différent, puis compile chaque fichier C++ en un binaire au format shared-object (`.so`) avec diverses options de compilation (on peut utiliser le compilateur gcc ou clang, et tester notamment l'usage de l'option `-march=native`.). Ce binaire est ensuite chargé par un programme écrit en C++ qui exécute sa boucle principale autant de fois que souhaité (1000 par défaut), en mesurant, pour chaque boucle, un ensemble d'évènements de la libPFM passés en paramètres. Les compteurs ainsi obtenus sont ensuite affichés sous forme de graphiques montrant leur évolution au cours du temps.

3.3.1 Modes d'exécution

Par défaut, les DSP sont exécutés *offline* : les signaux sont calculés mais ne sont pas joués en temps réel. Ceci peut poser problème dans les mesures d'utilisation du cache, car lorsqu'un signal est joué en temps réel, le processus chargé de le calculer n'est appelé que lorsqu'un nouveau buffer est nécessaire et est donc dormant la majorité du temps. Ceci diminue la probabilité de trouver les données mémoire au plus bas niveau de cache lorsque le processus est réveillé. Pour simuler des conditions de temps réel, on a plusieurs possibilités :

— Appeler la boucle principale avec de très grands buffers d'entrée/sortie (de l'ordre de 48000 échantillons) qui ne peuvent pas tenir dans le cache L1, pour s'assurer qu'ils ne

2. `https://perfmon2.sourceforge.net/manv4/libpfm.html`
3. `https://github.com/Minos/faust-scheduling-tools`

soient pas déjà dans le cache au début de la boucle : c'est la technique utilisée au chapitre 2 ;
— Au lieu d'utiliser toujours les mêmes buffers d'entrée/sortie entre chaque appel, on peut en utiliser des différents à chaque fois (c'est la méthode par défaut) ;
— Enfin, on peut tout simplement exécuter nos DSP en conditions réelles : le programme accepte les options `--alsa` et `--jack` pour exécuter le DSP mesuré directement dans, respectivement, ALSA [4] ou JACK [5]. Le temps réel augmente considérablement le temps nécessaire pour prendre une mesure mais produit en contrepartie le résultat le plus réaliste.

En mode *offline*, les buffers d'entrée sont remplis avec du bruit blanc. En conditions réelles, les entrées/sorties matérielles sont utilisées.

3.3.2 Graphiques de performances

Les évènements que l'on souhaite mesurer sont passés avec l'option `-e`, suivie d'une liste d'évènements séparés par des virgules. Les noms des évènements sont similaires à ceux acceptés par l'outil `perf` utilisé précédemment, mais diffèrent parfois des noms que l'on peut obtenir avec la commande `perf list`. Pour avoir la liste définitive des évènements supportés sur sa propre architecture, on peut exécuter la commande `pfm_info` (à la racine du projet). À titre d'exemple, on peut afficher le nombre de cycles, le nombre d'instructions et le nombre de succès et défauts de cache L1 pour un programme donné de la manière suivante :

```
fcschedtool plot mydsp.dsp \
        -e cycles,instructions,l1-dcache-loads,l1-dcache-load-misses
```

La plupart des PMU n'ont qu'une poignée de compteurs : ils sont au nombre de 4 sur Intel x86. Par défaut, lorsqu'on passe une liste de plus de 4 évènements à la libPFM (ou à la commande `perf`), le noyau Linux effectue une rotation entre les compteurs et on obtient des comptes incomplets. Pour éviter ce problème, on constitue des groupes de 4 évènements et chaque itération ne mesure qu'un groupe. On augmente le nombre d'itérations en conséquence pour que chaque groupe ait été mesuré le nombre de fois souhaité.

3.3.3 Tests de validité

La commande `fcschedtool test` permet de vérifier que les différentes stratégies d'ordonnancement d'un programme donné ont bien toutes le même comportement. Pour cela, on remplit les canaux d'entrée de chaque version du programme avec le même bruit blanc (la graine aléatoire utilisée pour le générer est toujours la même), et on compare les canaux de sortie. Pour éviter les erreurs d'arrondi, le code C++ est compilé sans l'option `-ffast-math` et au niveau 0 d'optimisation (`-O0`).

On peut lui passer en argument une liste de programmes Faust, ou un dossier contenant des programmes Faust, pour vérifier rapidement les stratégies d'ordonnancement sur un ensemble de programmes.

3.4 Observations

Les mesures qui suivent se sont focalisées sur 22 programmes Faust, issus des exemples, pour lesquels on avait mesuré un rapport de temps de calcul supérieur à 1,5 entre l'ordonnancement le moins efficace et l'ordonnancement le plus efficace.

4. Advanced Linux Sound Architecture : `https://www.alsa-project.org/`
5. JACK Audio Connection Kit : `https://github.com/jackaudio/jack2`

3.4.1 Quantité d'instructions et pipeline stalls

Les premiers résultats significatifs ont été obtenus en affichant au cours du temps les quatre évènements suivants, mesurés entre le début et la fin d'une exécution de la boucle principale, pendant 1000 itérations :

— `cpu-cycles` : le nombre de cycles CPU écoulés ;
— `instructions` : le nombre total d'instructions exécutées ;
— `cycle_activity.stalls_mem_any` : le nombre de cycles passés sans exécuter d'instructions dans l'attente d'une donnée mémoire, quelque soit le niveau de cache dans lequel elle se trouve ;
— `cycle_activity.stalls_total` : le nombre de cycles passés sans exécuter d'instructions, quelle que soit la raison. En retranchant à ce nombre la mesure précédente, on obtient l'évènement `stalls_other`.

Pour mesurer l'impact du compilateur C++ utilisé, ces figures ont été générées avec l'option `--exhaustive` : chaque programme est alors testé avec Clang et avec GCC, et avec ou sans l'option `-march=native`.

Les évènements `cycle_activity.stalls_mem_any` et `cycle_activity.stalls_total` sont spécifiques à l'architecture Intel SkyLake, ce qui ne garantie pas la possibilité d'obtenir des figures équivalentes sur une architecture différente. Pour augmenter la lisibilité des résultats, les instructions sont représentées divisées par 4 par rapport aux autres mesures, ce qui correspond au minimum théorique du nombre de cycles nécessaire à leur évaluation sur un processeur superscalaire de taille 4. Les stalls pipeline sont additionnés à ce nombre, et la somme totale est toujours inférieure au égale au nombre de cycles.

L'option `-p stalls` de `fcschedtool plot` permet de reproduire ces mesures et de tracer des figures similaires. Il est également possible de donner en argument un dossier contenant des programmes Faust, et l'option `-o <output_dir>` permet de spécifier un dossier dans lequel exporter les figures. La ligne de commande complète utilisée pour tracer les graphiques qui suivent est donc :

```
fcschedtool plot <dsp_directory> -p stalls --exhaustive -o <output_directory>
```

Les figures 3.4, 3.5 et 3.6 montrent des exemples caractéristiques des résultats obtenus. La figure 3.4 montre un cas idéal, mesuré sur un programme très simple utilisant très peu de mémoire. Dans ce cas, le nombre de cycles d'exécution est proche du quart du nombre d'instructions exécutées, et il n'y a presque aucun cycle perdu à attendre des données. Ce programme a surtout servi à valider la méthode : il ne fait pas partie des programmes problématiques.

Dans ces figures, on n'affiche pas le temps d'exécution en millisecondes car celui-ci est directement proportionnel au nombre de cycles : sur un CPU cadencé à 3 gigahertz, 3 milliards de cycles s'exécutent en une seconde. L'usage du cycle comme unité de temps permet de mieux relier la vitesse d'exécution aux compteurs d'évènements qui sont du même ordre de grandeur. Il faut cependant s'assurer que la fréquence du processeur reste constante pendant toute la mesure : si les temps de calculs sont directement proportionnels au nombre de cycles, ce n'est pas le cas des latences des différents niveaux de mémoire qui sont fixes quelque soit la cadence du processeur.

Pour les programmes qui nous intéressent, deux tendances semblent se dégager : pour 12 programmes sur 22, le nombre de cycles CPU varie avec la quantité d'instructions exécutées (exemple : le programme `vocoder` représenté figure 3.5), et pour 9 programmes sur 22, la différence semble liée au nombre de cycles passés en attente de données (exemple : le programme `crazyGuiro` représenté figure 3.6).

Cette dernière figure fait apparaître des pics périodiques dans le nombre de cycles d'attente. Ce motif apparait fréquemment dans des programmes utilisant des lignes à retard, et semble liés à l'utilisation de buffers circulaires : la plupart des boucles n'accèdent qu'à un seul segment contigu

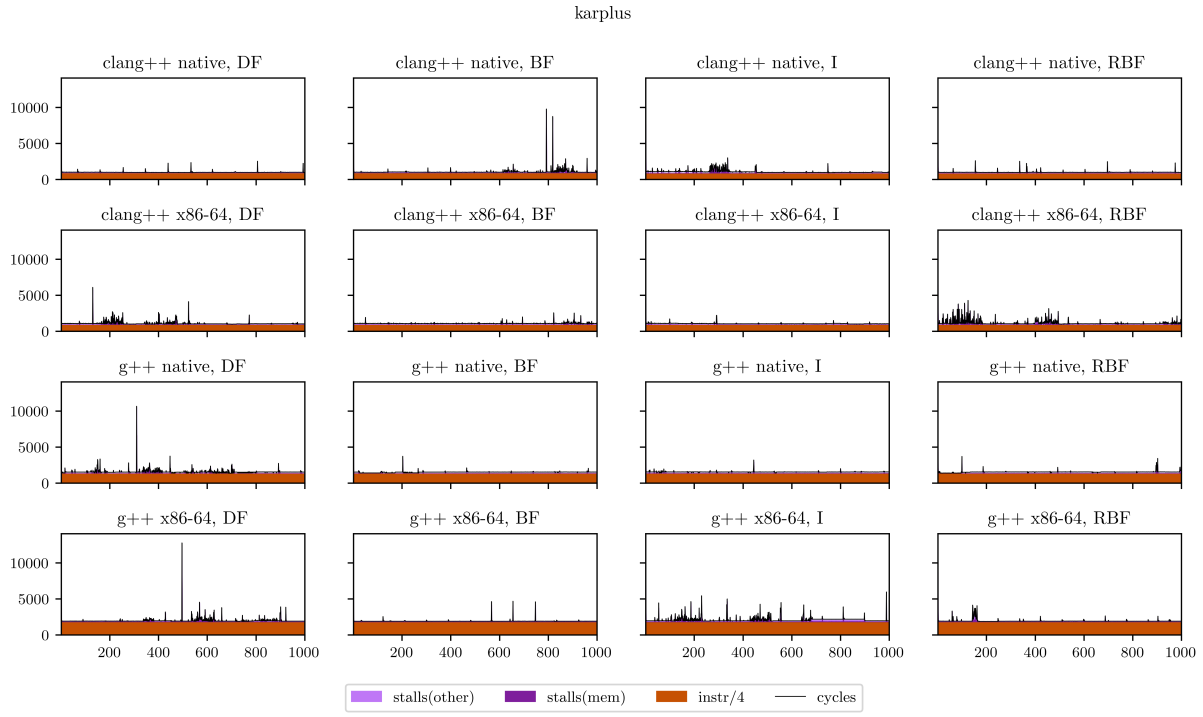

FIGURE 3.4 – Mesures détaillées pour le programme karplus. Le nombre d'instructions (en orange) est représenté divisé par 4 pour mettre en évidence que le nombre de cycles (trait noir) correspond au quart du nombre d'instructions exécutées pour produire une seconde de signal.

du buffer circulaire, mais ont besoin périodiquement de deux segments discontinus lorsque la fin du buffer est atteinte. Ce motif reste stable d'une mesure à l'autre, et change lorsqu'on modifie la longueur du délai à l'origine de la ligne à retard dans le code Faust.

On remarque aussi que sur le programme `vocoder`, les différences qu'on avait observées avec Clang ne se produisent pas avec GCC, qui produit quasiment les mêmes performances pour toutes les stratégies, bien que celles-ci soient presque toujours moins bonnes que celles de Clang. De manière générale, les performances des programmes compilés avec Clang sont meilleures qu'avec GCC, mais au prix d'une plus grande variabilité en fonction de l'ordonnancement.

L'ensemble des figures obtenues est consultable à l'adresse suivante : `https://simon.jacquin.me/faust.html`.

Ces mesures mettent en évidence une disparité entre les quatre stratégies d'ordonnancement : les parcours en largeur semblent généralement mieux optimisés par les compilateurs car il s'exécutent avec moins d'instructions en moyenne, mais ils produisent des effets de cache plus importants. Pour confirmer cette tendance, on s'attache dans la section suivante à mesurer un maximum d'évènements différents, pour observer d'une part l'efficacité du superscaling (quantité d'instructions par cycle) et de la vectorisation, et d'autre part les défauts de cache.

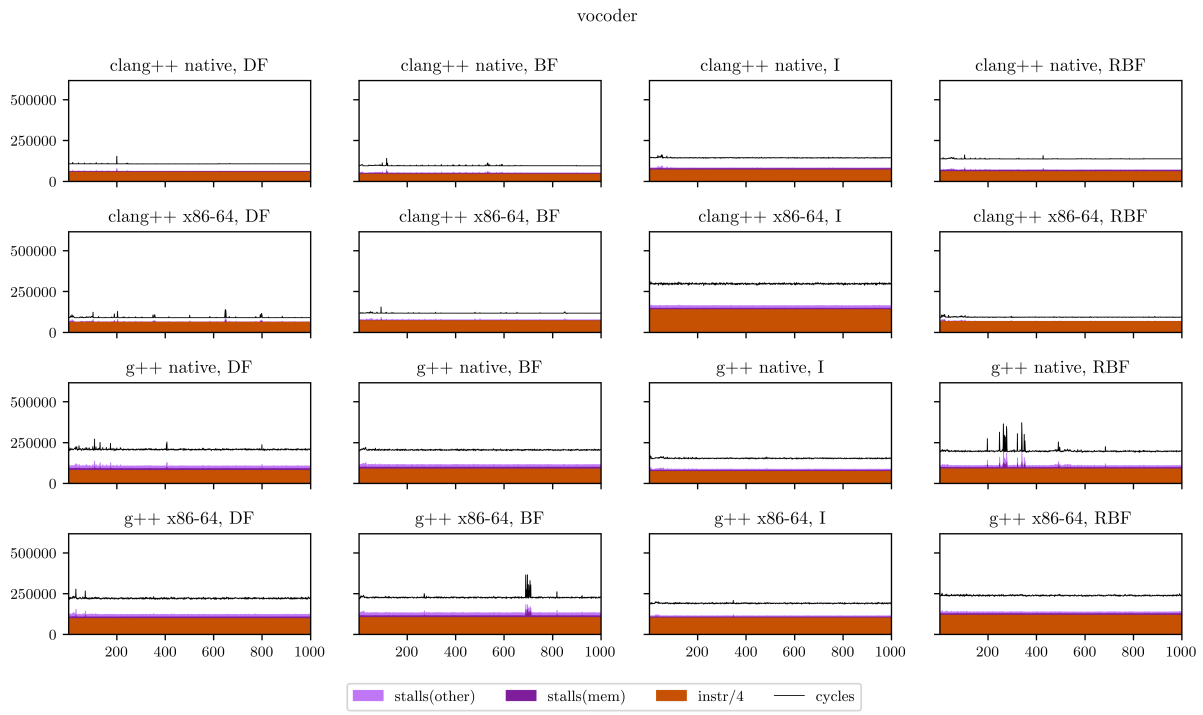

FIGURE 3.5 – Mesures détaillées pour le programme vocoder. On voit que l'écart de performances correspond ici à une différence dans la quantité d'instructions exécutées, et que la variabilité observée avec clang n'apparait pas avec gcc.

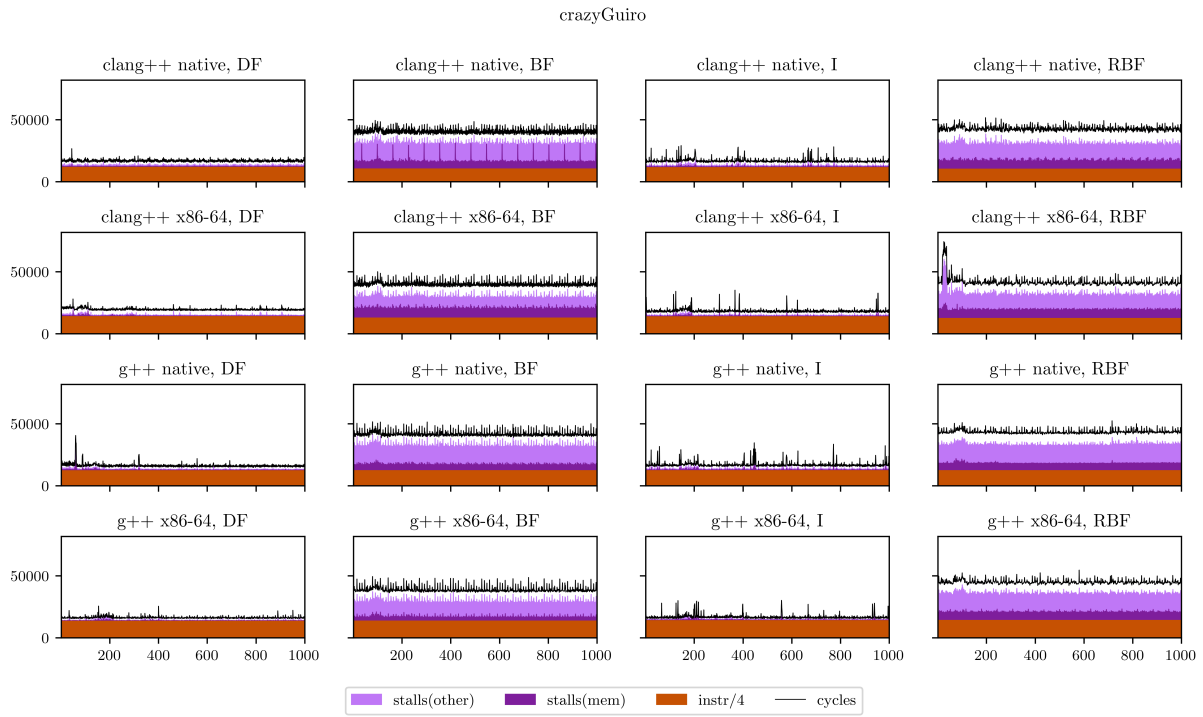

FIGURE 3.6 – Mesures détaillées pour le programme crazyGuiro. Les *pipeline stalls* sont représentés en violet et additionnés aux instructions. Ici, l'écart de performances entre les quatre stratégies est systématique quelque soit le compilateur C++ utilisé.

3.4.2 Bilan d'utilisation des ressources

Les figures obtenues précédemment montrent un bruit de mesure assez faible, aussi pour pouvoir comparer davantage de données à la fois, on ne conservera pour chaque mesure que la valeur médiane sur 100 itérations de la boucle principale.

La sous-commande `fcschedtool summary` évalue, pour chaque programme d'entrée, trois groupes d'évènements :

1. Les évènements qui rendent compte de l'occupation du pipeline : `uops_executed.cycles_ge_{1,2,3,4}` comptent le nombre de cycles pendant lesquels respectivement au moins 1, 2, 3 ou 4 micro-opérations sont exécutées, et `cycle_activity.stalls_mem_any` et `cycle_activity.stalls_total` mesurent, comme précédemment, les cycles d'attente du pipeline.
2. Les évènements qui exposent la vectorisation : `fp_arith.scalar_single` correspond à des opérations flottantent non-vectorisées, et `fp_arith.{128b,256b}_packed_single` à des opérations vectorisées par respectivement 4 et 8.
3. Les évènements qui mesurent l'utilisation de la mémoire : `l1-dcache-loads` et `l1-dcache-load-misses` mesurent les succès et défauts du cache L1 en lecture, `l1-dcache-stores` et `l1-dcache-store-misses` en écriture. On entend par lecture la copie d'une valeur en mémoire vers un registre, et par écriture l'opération inverse.

Dans les figures 3.7 et suivantes, ces évènements sont affichés par groupe. Dans le groupe *cycles*, on affiche la répartition du nombre d'opérations exécutées par cycle, entre 4 et 0 (stalls). L'encadré noir représente le nombre total de cycles : il rend compte de la performance globale, et il est directement proportionnel au temps d'exécution. Dans le groupe *flops*, le total représente le nombre d'opérations flottantes exécutées, décomposées entre les opérations vectorisées par 8 (rouge foncé), par 4 (rouge clair) et non vectorisées (jaune). Enfin, le groupe *memory* montre les accès à la mémoire : les lectures et écritures sont représentées en bleu clair lorsqu'elles se font dans le cache L1 (cache-hit), et en bleu foncé lorsqu'elles se font en dehors (cache-miss). Dans tous les cas, l'axe des abscisses représente le nombre d'occurrences de l'évènement correspondant au cours d'une exécution de la boucle principale.

Les programmes n'utilisant ni canaux d'entrée ni lignes à retard ont peu d'empreinte mémoire et semblent favoriser les parcours en largeur : c'est le cas du programme drums.dsp représenté figure 3.7. Les parcours en largeur (BF et RBF) présentent deux avantages notables : la moitié des opérations flottantes sont vectorisées, et la quantité totale d'accès mémoire est inférieure, ce qui signifie que davantage de calculs sont effectués directement dans les registres. On économise ainsi des cycles d'attente ainsi que des micro-opérations destinées à sauvegarder puis récupérer les calculs.

Les programmes ayant une empreinte mémoire plus importante, à l'inverse, sont plus performants avec un parcours en profondeur (DF) ou entrelacé (I) : le cas du programme freeverb (figure 3.8) est emblématique. Bien que les parcours en largeur favorisent toujours la vectorisation et une meilleure allocation de registres, cet avantage est perdu car le nombre de défauts de cache est multiplié par 5, ce qui entraîne un bien plus grand nombre de cycles d'attente.

Dans ce programme, on doit aussi noter que la plupart des cycles d'attente ne sont pas des cycles d'attente mémoire. Un autre mécanisme engendrant des cycles d'attente pourrait expliquer ce phénomène : lorsque toutes les instructions qui pourraient être ajoutées au pipeline dépendent de résultats qui sont en cours de calcul, le processeur est obligé de perdre des cycles en attendant que ces calculs soient terminés : on parle de *backend stalls*. La plupart des programmes dans lesquels ce pattern apparait produisent également beaucoup de tels cycles d'attente.

Cependant, si on serait tenté de croire que les parcours en largeur favorisent systématiquement l'allocation de registres, on constate que ce n'est pas toujours le cas. Dans la figure 3.9, on voit

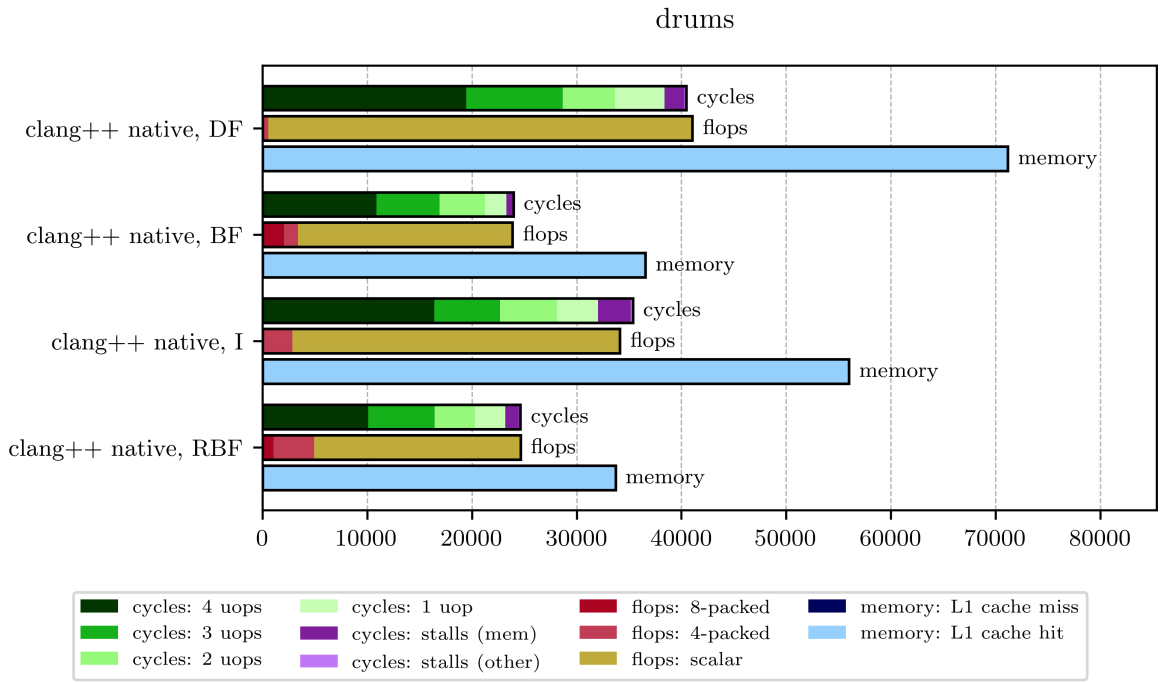

FIGURE 3.7 – Bilan du programme drums.dsp. Dans les parcours en largeur, la moitié des opérations flottantes (flops) sont vectorisées par 4 ou par 8, ce qui semble expliquer le gain de performances.

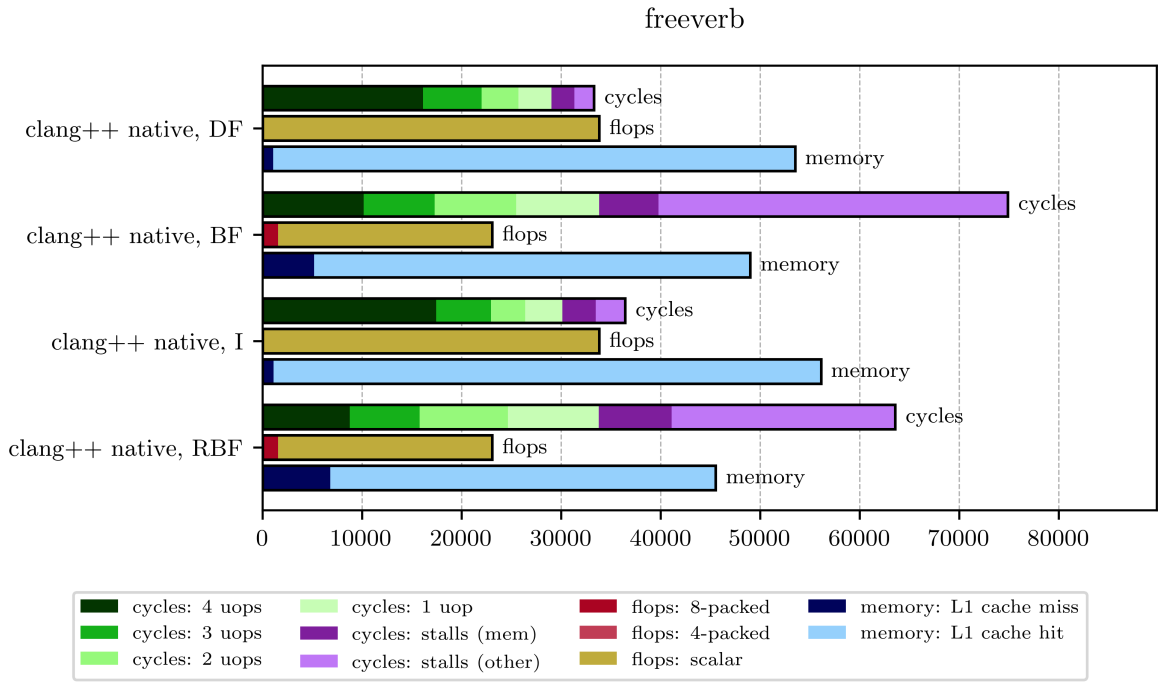

FIGURE 3.8 – Bilan du programme freeverb.dsp. Comme pour le programme drums.dsp, les parcours en largeur favorisent la vectorisation. Cependant, les accès mémoire sont moins bien localisés et engendrent davantage de défauts de cache, ce qui augmente la quantité de cycles d'attente.

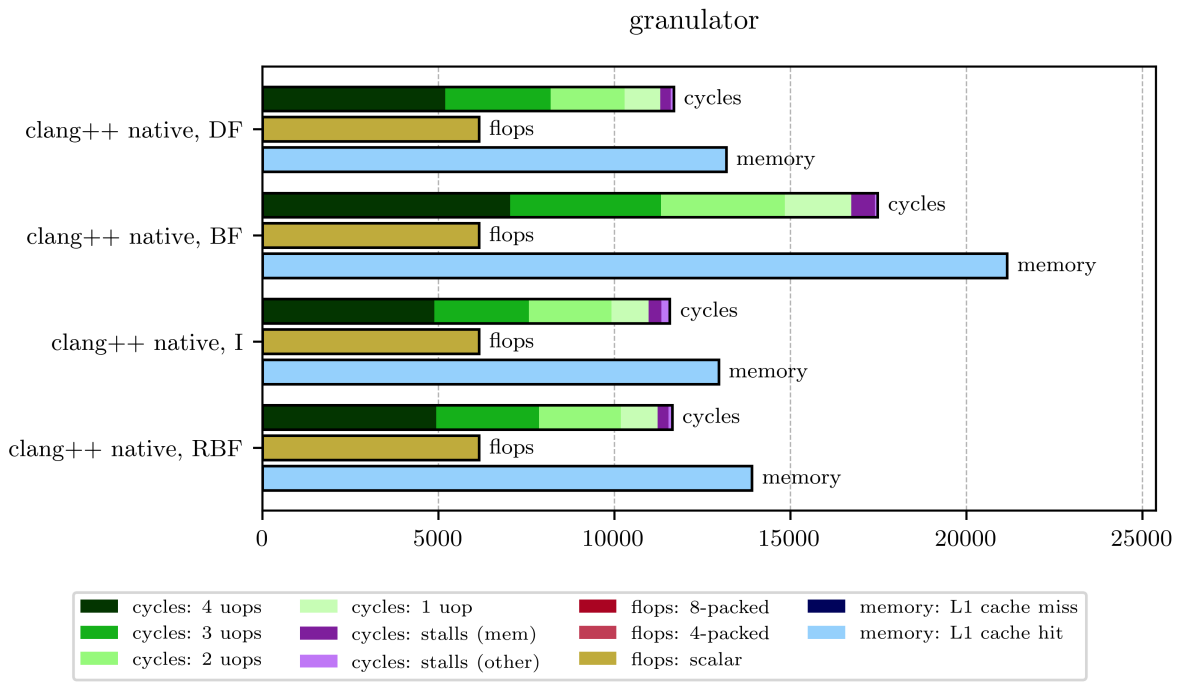

FIGURE 3.9 – Dans le programme granulator.dsp, le nombre de flops semble rigoureusement le même entre les quatre stratégies. Cependant, le parcours en largeur (BF) est significativement plus lent que les autres : ceci s'explique ici par un plus grand nombre d'accès mémoire, le compilateur n'ayant pas réussi à faire tenir autant de calculs dans les registres (*register spill*).

que toutes choses étant égales par ailleurs, le parcours en largeur normal a bien plus recours à la mémoire que les autres, bien que le parcours en largeur inversé, lui, ne soit pas affecté.

On observe également des résultats qui dépendent plus directement du comportement du compilateur C++ : dans la figure 3.10, la stratégie *interleaved* exécute davantage d'opérations flottantes que les parcours en largeur, alors que celles-ci sont moins vectorisées. En effet, outre la vectorisation, les compilateurs peuvent également générer opportunément des instructions FMA (fused multiply-add) et ainsi fusionner ensemble deux opérations flottantes.

Enfin, la figure 3.11 montre un cas plus chaotique : le nombre d'opérations flottantes varie entre toutes les stratégies, et le parcours en largeur (BF) a l'empreinte mémoire la plus faible, quand le parcours en largeur inversé (RBF) a l'empreinte mémoire la plus forte.

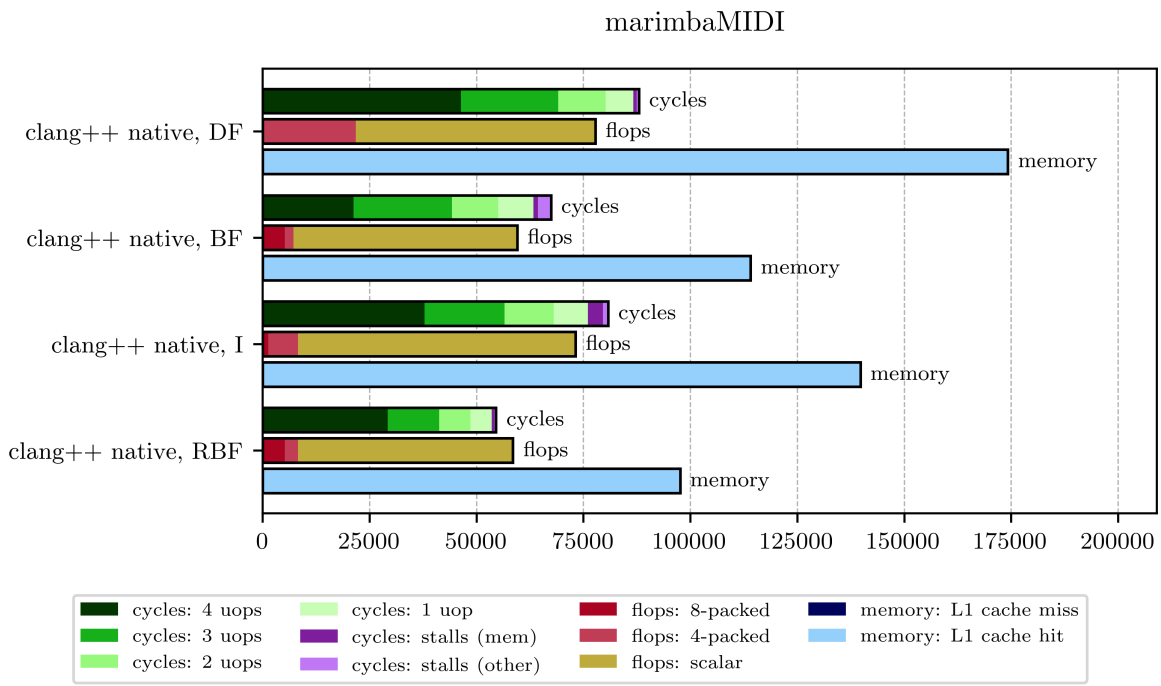

FIGURE 3.10 – Dans le programme marimbaMIDI.dsp, le nombre total d'opérations flottantes est différent entre la première stratégie et les trois autres : le compilateur Clang n'a pas pu trouver les mêmes optimisations. Le reste des différences s'explique par la vectorisation et les allocations de registres.

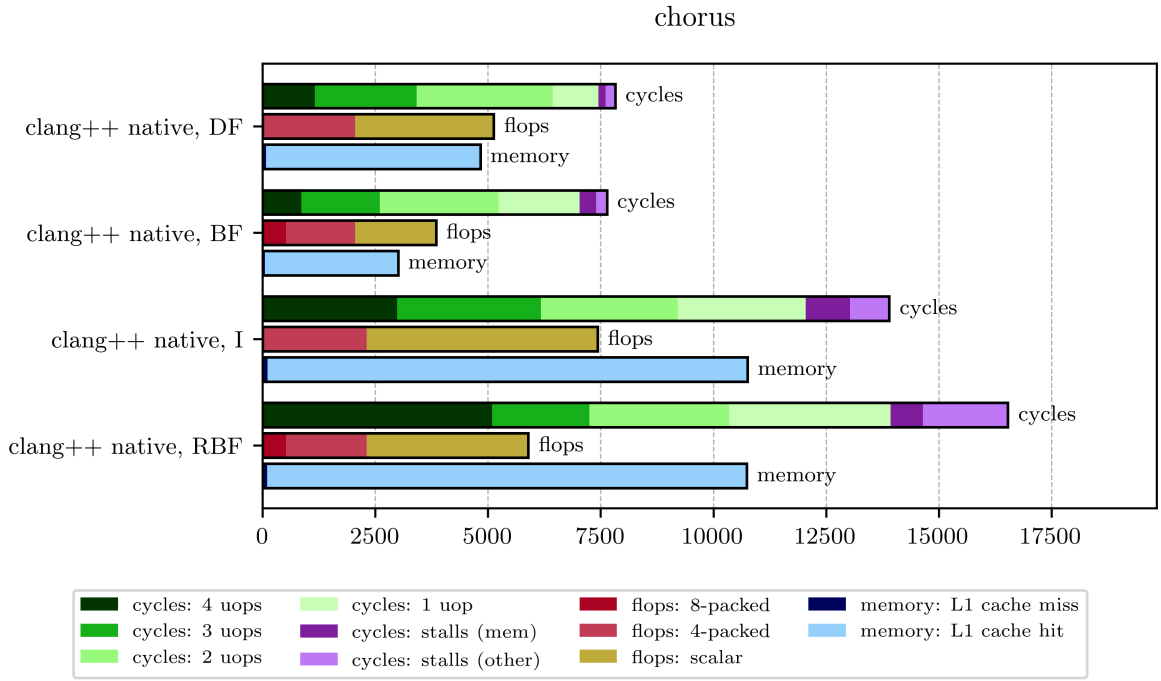

FIGURE 3.11 – Bilan du programm chorus.dsp. Les parcours BF et RBF produisent une meilleure vectorisation, et les parcours DF et BF ont 2 à 4 fois moins d'empreinte sur la mémoire que les deux autres. Le spill de registres du parcours DF s'équilibre avec la vectorisation du parcours BF dans cette mesure, mais on peut penser qu'une machine différente produirait d'autres résultats.

3.5 Conclusion

Si ce chapitre ne nous a toujours pas permis de trouver la meilleure stratégie d'ordonnancement pour générer du code C++ performant, on commence cependant à comprendre les phénomènes qui engendrent les comportements en apparence chaotiques des différents algorithmes de tri topologique.

1. **Autovectorisation** : lorsque plusieurs opérations flottantes similaires et indépendantes se suivent, les compilateurs sont capables de les regrouper en une seule instruction SIMD, permettant ainsi d'effectuer 4, 8, voire 16 calculs en une seule instruction. Les parcours en largeur favorisent l'autovectorisation car ils placent les unes à la suite des autres des opérations indépendantes. Les parcours entrelacés semblent moins efficaces, et les parcours en profondeur créent des chaînes d'opérations dépendantes les unes des autres, peu propices à l'autovectorisation.
2. **Allocation de registres** : lorsque trop de calculs doivent être retenus en mémoire simultanément, le compilateur ne peut plus se contenter de générer des instructions de calcul sur les registres. Il doit ajouter des instructions pour sauvegarder les résultats d'un calcul dans la mémoire, puis pour les récupérer plus tard : on parle de *register spill*. L'ordonnancement des instructions a un impact significatif ici, et les parcours en largeur semblent encore une fois favoriser une meilleure utilisation des registres.
3. **Défauts de cache** : lorsque les accès mémoire sont mal localisés, l'ensemble des données nécessaires à la production d'un seul échantillon ne tient plus dans le niveau L1 du cache et on peut perdre beaucoup de cycles à réaffecter constamment des lignes de cache. Les parcours en profondeur semblent plus adaptés dans ce genre de cas, puisque les résultats ont tendance à être utilisés immédiatement après leur calcul, ce qui peut éviter de charger plusieurs fois la même ligne de cache au cours de la même itération.

Chapitre 4

Améliorations

4.1 Le tri topologique dans la génération de code assembleur

Le problème de l'ordonnancement des instructions par les compilateurs est étudié au moins depuis les années 1970 [5]. Le besoin s'est manifesté avec le développement des techniques d'*instruction level parallelism* et de *pipelining* développées au chapitre 3. Comme le compilateur Faust, les compilateurs modernes utilisent une représentation par un graphe orienté acyclique de dépendances entre les instructions à exécuter et effectuent un tri topologique de ce graphe pour obtenir une séquence valide [6].

L'algorithme dominant depuis les années 1970 est appelé *List Scheduling* [6] [5] [7]. Il s'agit d'un algorithme glouton, de conception différente des techniques testées jusqu'ici par le compilateur Faust. Il procède de la manière suivante :

1. On construit le graphe D des dépendances entre les instructions. Chaque sommet du graphe est annoté avec la latence de l'instruction correspondante, en nombre de cycles.
2. On assigne à chaque instruction une priorité, qui sera utilisée pour guider l'ordonnancement. De nombreux systèmes de priorités existent ; un schéma classique consiste à utiliser la longueur du plus long chemin, pondéré par la latence des opérations, entre le sommet et une racine de D.
3. On sélectionne itérativement un sommet dans le graphe qu'on ajoute à la séquence S d'instructions. Ce sommet est sélectionné parmi tous les sommets dont l'ensemble des dépendances se trouvent déjà dans la séquence S.

L'algorithme maintient deux listes d'instructions : une liste *Ready* qui contient les instructions dont toutes les opérandes sont déjà présentes dans la séquence S et dont le calcul est terminé (on compare pour cela la latence de l'instruction à la somme des latences des instructions qui la suivent dans la séquence S), et une liste *Active* qui contient les instructions présentes dans la séquence S mais dont le calcul n'est pas encore terminé. Chaque itération met à jour les deux listes.

Le mécanisme de sélection d'une instruction à ajouter à la séquence constitue le cœur de cet algorithme. En plus du plus long chemin pondéré par la latence des opérations évoqué précédemment, on peut prendre en compte :

— le nombre de successeurs immédiats du nœud dans D : en insérant plus tôt un nœud dont dépendent beaucoup d'autres nœuds, on augmente la taille de la liste *Ready*, ce qui augmente la marge de manœuvre de l'algorithme ;

— le nombre total de descendants du nœud dans D, selon la même idée ;

— la latence de l'instruction : en plaçant plus tôt une opération longue, on augmente la quantité d'opérations indépendantes qui peuvent être placées après pour couvrir sa latence ;

— le nombre de calculs effectués précédemment qui seront utilisés pour la dernière fois par l'opération considérée : ceci permet d'augmenter les chances de trouver le résultat de ces opérations dans un registre ou dans un cache de bas niveau.

La littérature sur cet algorithme se concentre principalement sur la génération d'instructions assembleur. Pour l'adapter au compilateur Faust, il peut être intéressant d'étudier d'autres mécanismes de sélection plus adaptés aux spécificités des programmes Faust : ceux-ci contiennent généralement peu de branches, mais requièrent de nombreux accès mémoire à des zones contigües, ce qui les rend particulièrement sensibles aux effets de cache. De plus, le code généré est appelé à être lui-même compilé et réordonné, ce qui complique la possibilité de prédire son comportement.

4.2 Un List Scheduling pour Faust

On a vu dans le chapitre précédent que les parcours en largeur obtiennent de meilleures performances en auto-vectorisation et en allocation de registres, mais deviennent mauvais lorsque l'ensemble des données d'une itération ne tient pas dans le plus bas niveau de cache. Pour comprendre concrètement cet effet, intéressons nous de plus près au code généré pour le programme freeverb.dsp, dont on a analysé les performances figure 3.8.

Ce programme est implémenté en utilisant, notamment, 16 buffers circulaires de 8192 flottants qui sont utilisés comme lignes à retard. Ceci représente 512 Kio de données, soit bien plus que la taille typique du niveau L1 du cache (cf. tableau 3.3). Avec la stratégie *deep-first*, à chaque itération, on observe 16 motifs de la forme suivante :

```
fVec3SE[IOTA0&8191] = ((fSlow5BE * fVec1SE[0]) + fTemp2SE);
fVec0SE[0] = fVec3SE[(IOTA0-iConst10KI)&8191];
```

On voit qu'ici, la première ligne écrit dans le buffer `fVec3SE` puis la seconde lit dans le même buffer à une position antérieure. Étant donné que les 16 buffers sont positionnés immédiatement les uns après les autres dans la structure de données du DSP, et qu'ils sont écrits puis lus dans l'ordre dans lequel ils ont été déclarés, il est probable que chaque buffer n'ait besoin d'être chargé qu'une seule fois lors d'une itération.

En revanche, avec la stratégie *breadth-first*, on voit d'abord 16 lignes qui écrivent chacune dans un buffer circulaire, puis un peu plus loin, 16 lignes qui lisent dans ces buffers. Les 16 lectures ayant probablement rempli le niveau L1 du cache, les 16 écritures suivantes risquent d'aboutir à nouveau à un défaut de cache. Chaque buffer sera alors chargé deux fois par itération.

En analysant le code des programmes pour lesquels on observe de nombreux défauts de cache, on observe que tous utilisent de nombreuses lignes à retard qui ne peuvent pas toutes tenir en même temps dans le plus bas niveau de cache.

On propose donc un nouvel algorithme d'ordonnancement basé sur le List Scheduling. Pour imiter le comportement d'un parcours en largeur, on affecte à chaque instruction une latence de 1 : tout autre système de latence serait compliqué à mettre en œuvre étant donné qu'on ne connait pas l'architecture sur laquelle le programme sera finalement compilé. Ainsi, la section critique de chaque sommet devient égale au nombre de sommets qui le séparent de sa racine la plus éloignée. En utilisant ce seul critère de priorité, on obtiendrait un parcours en largeur inversé : tous les sommets ayant la même section critique seraient planifiés ensemble.

Pour améliorer ce parcours, on ajoute un autre système de priorité : lorsqu'une instruction accédant à une ligne à retard est planifiée, toutes les instructions qui accèdent à la même ligne à retard deviennent prioritaires, quelle que soit leur section critique.

Le système de priorité devient alors, par ordre d'importance :

1. En premier, choisir un sommet qui accède à une ligne à retard récemment utilisée ;

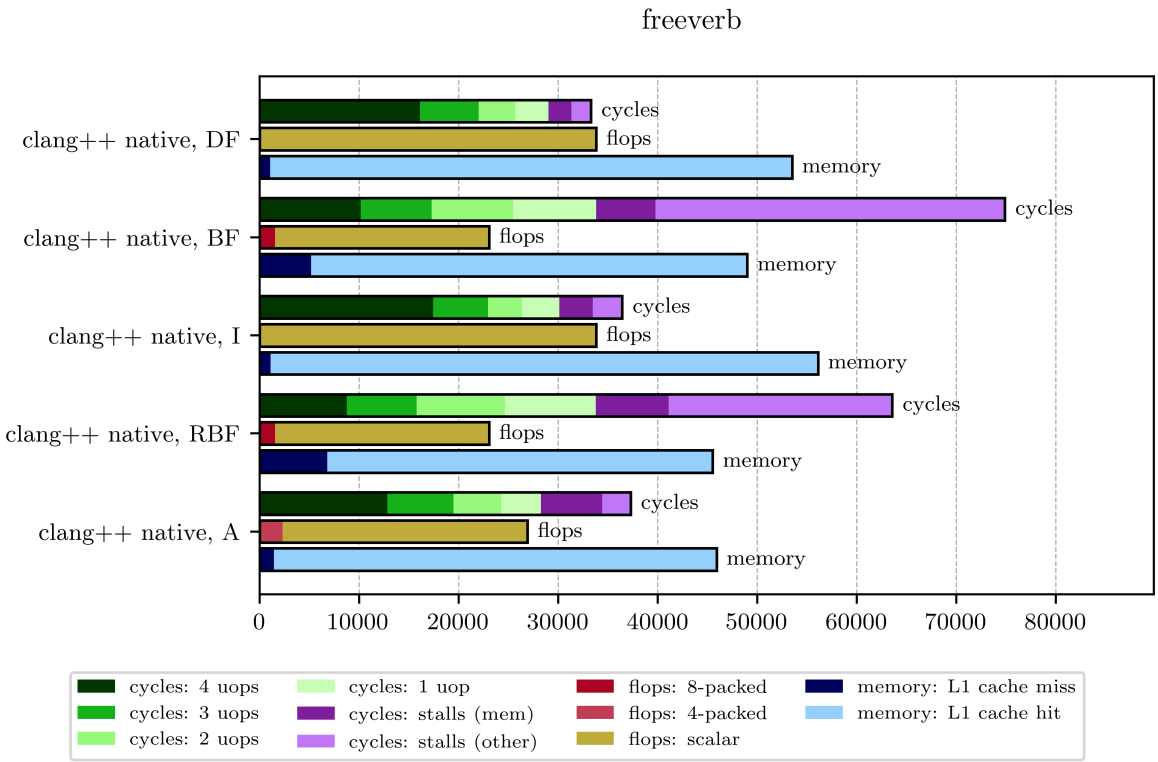

FIGURE 4.1 – La stratégie *adaptive* réussit à mieux vectoriser les opérations flottantes que les stratégies *deep-first* et *interleaved*, tout en évitant de multiplier les défauts de cache.

2. Si aucun ou plusieurs sommets correspondent à ce critère, on sélectionne celui qui a la plus longue section critique.
3. Si plusieurs sommets ont la même section critique, on favorise ceux dont un maximum de dépendances seront lues pour la dernière fois.

Cette nouvelle stratégie, appelée *adaptive*, est implémentée dans une branche expérimentale du compilateur, accessible à l'adresse suivante : `https://github.com/Minos/faust`.

Pour une majorité des programmes de test, les résultats sont plutôt favorables : si les performances des parcours en largeur sont conservées avec cette nouvelle stratégie, on parvient à éviter des défauts de cache et à rattraper les temps des parcours en profondeur sur certains programmes utilisant des lignes à retard comme le programme freeverb.dsp représenté figure 4.1.

Les figures 4.2 et 4.3 montrent la performance de cette nouvelle stratégie par rapport aux quatre stratégies préexistantes. Malgré quelques cas critiques (notamment les programmes chorus et zitaRev), elle parvient à se situer dans la moyenne basse. Notre nouvelle stratégie s'exécute en moyenne 10% plus vite que la moyenne sur notre corpus de tests, et 4% plus vite sur celui de la figure 4.3, avec une déviation standard de l'ordre de 20%. Il s'agit, dans les deux cas, de la meilleure performance, même si cette mesure semble peu significative au regard de la grande variabilité des temps mesurés.

4.3 Mode vectoriel et séparation des boucles

Depuis le début de ces expériences, on s'est focalisés uniquement sur le mode scalaire de Faust. On s'est donc posé une contrainte : dans ce mode de fonctionnement, les signaux sont générés par une seule boucle qui réalise l'ensemble des calculs échantillon par échantillon, comme représenté figure 4.4.

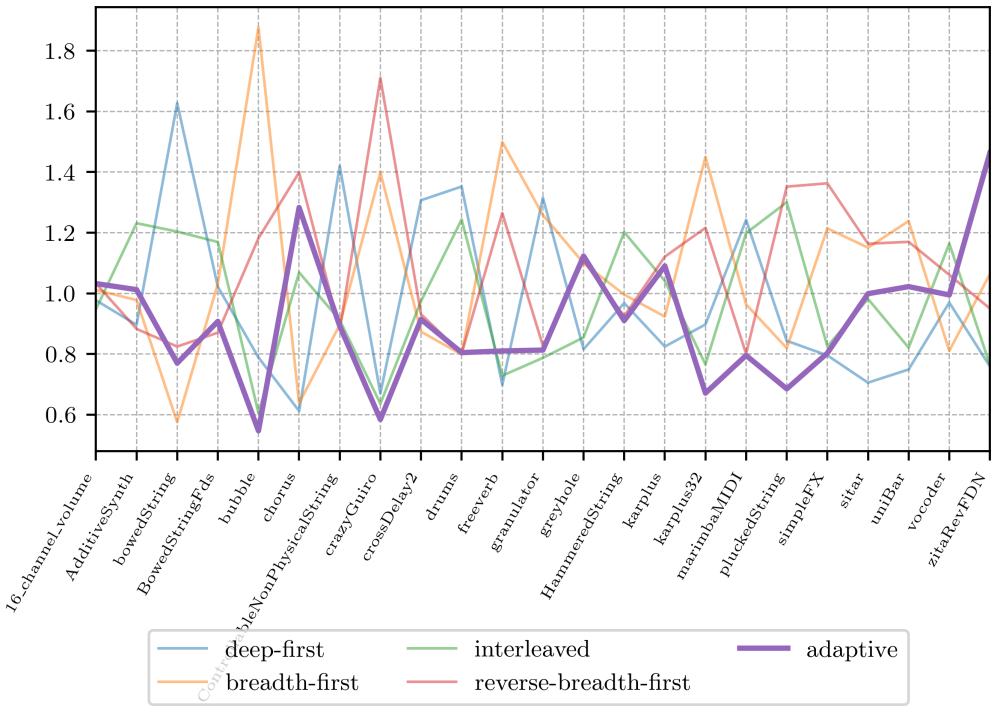

FIGURE 4.2 – Temps d'exécution de chaque stratégie par rapport au temps moyen pour les 23 programmes étudiés. La stratégie *adaptive* est représentée avec un trait plus épais.

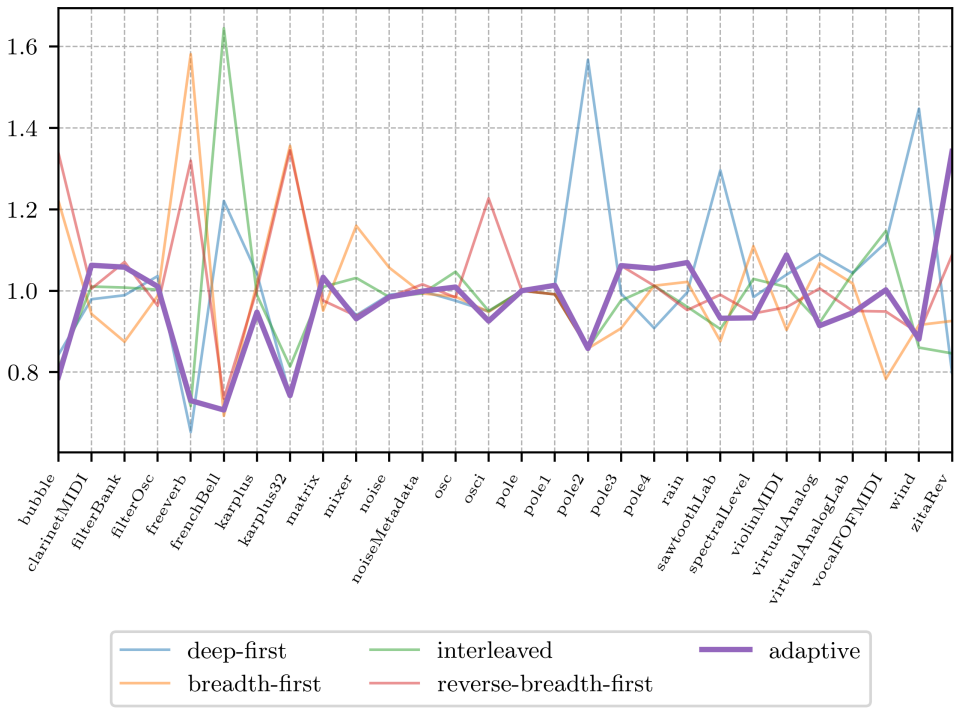

FIGURE 4.3 – Temps d'exécution de chaque stratégie par rapport au temps moyen pour les programmes testés dans l'article de Orlarey, Letz, Fober et Michon (2020) [2].

```
void compute(int count, float** input, float** output)
{
    /* slow computations, initializations, pre-processing */
    for (int i=0; i<count; i++) {
        /* sample-by-sample computations on all buffers */
    }
    /* post-processing */
}
```

FIGURE 4.4 – Schéma du mode scalaire de Faust

Ce mode de fonctionnement présente plusieurs inconvénients. D'une part, la longueur de la boucle principale, qui peut dépasser plusieurs milliers de lignes dans nos exemples, rend beaucoup plus difficile l'autovectorisation et les optimisations du compilateur C++. D'autre part, on a vu au chapitre 3 que la mémoire du CPU est organisée pour des accès localisés en temps et en espace. Or dans ce format de boucle, on accède à chaque itération à l'ensemble des données du DSP, et ce quelque soit l'ordre des instructions dans cette boucle.

Le compilateur Faust implémente un autre mode de génération de code qui sépare le calcul des signaux en plusieurs boucles : le mode vectoriel [1]. Dans ce cas, on part du graphe de calcul complet (avec cycles) du programme, puis on créé un nouveau graphe en regroupant les cycles dans des super-nœuds : on obtient un graphe orienté acyclique. Chaque nœud de ce nouveau graphe peut alors être calculé dans une boucle séparée, et on utilise des vecteurs pour passer les résultats intermédiaires d'une boucle à l'autre. Dans ce mode, la localité des accès à la mémoire est mieux respectée et les boucles, plus simples, sont mieux vectorisées par les compilateurs.

Cependant, la multiplication des boucles et l'augmentation de l'espace en mémoire nécessaire pour passer les résultats d'une boucle à l'autre peuvent s'avérer contre-productives, et les performances obtenues peuvent être bien pires qu'en mode scalaire pour certains programmes. De plus, les avancées en matière d'autovectorisation dans les compilateurs de ces 15 dernières années ont, comme on a pu le voir, rendu le mode scalaire capable de générer du code vectorisé.

Si l'on ne cherche plus à produire du code facilement autovectorisable, mais simplement à maximiser la localité des accès à la mémoire, on peut cependant s'inspirer du mode vectoriel pour améliorer les avancées obtenues grâce à la stratégie étudiée précédemment : en conservant le mode scalaire pour tous les calculs n'utilisant pas d'entrées/sorties ni de lignes à retard, mais en isolant dans des boucles séparées les calculs qui accèdent à des buffers, on obtient alors des boucles dont on peut s'assurer que les données de calcul tiennent dans le cache.

En modifiant manuellement le code généré par la stratégie *adaptive* selon ce principe, on a pu diviser par 4 le temps de calcul du programme karplus32 : la figure 4.5 montre dans le détail les performances obtenues. La quantité d'accès mémoire est divisée par 20, ce qui signifie qu'on a pas seulement optimisé l'utilisation du cache, mais également l'utilisation des registres.

Cette approche demande cependant un travail important sur le compilateur. Si l'on considère un exemple simple qui consisterait à appliquer deux fonctions sur un signal d'entrée puis les additionner, une implémentation scalaire ressemblerait à ceci :

```
void compute(int count, float** input, float** output) {
    for (int i = 0; i < count; i++) {
        output[0][i] = f1(input[0][i]) + f2(input[0][i]);
    }
}
```

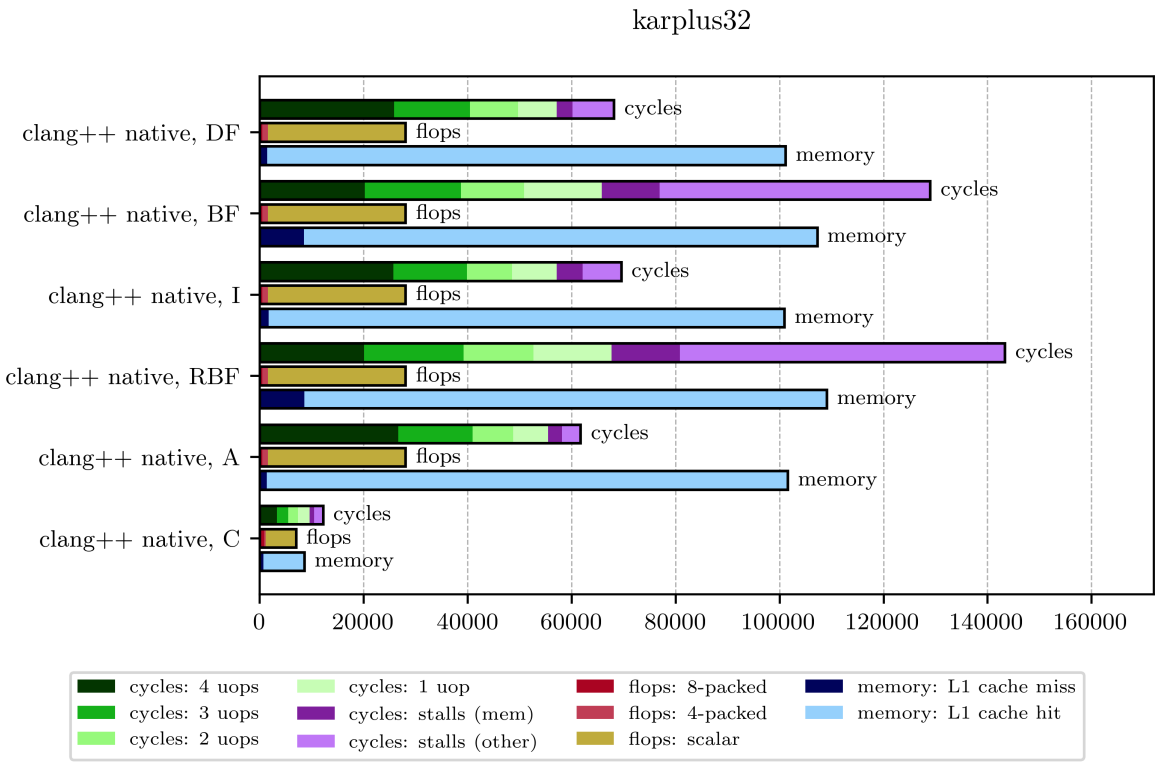

FIGURE 4.5 – La dernière stratégie (C) montre le comportement du programme karplus32 lorsqu'on place chaque ligne à retard dans une boucle séparée.

Si les fonctions `f1` et `f2` ont une forte empreinte mémoire et qu'on souhaite les séparer, dans le mode vectoriel de Faust, les résultats seraient stockés dans des vecteurs intermédiaires et l'implémentation utiliserait une boucle pour chaque fonction, puis une boucle pour l'addition :

```
void compute(int count, float** input, float** output) {
    float fTemp0SE[count];
    float fTemp1SE[count];

    for (int i = 0; i < count; i++) {
        fTemp0SE[i] = f1(input[0][i]);
    }

    for (int i = 0; i < count; i++) {
        fTemp1SE[i] = f2(input[0][i]);
    }

    for (int i = 0; i < count; i++) {
        output[0][i] = fTemp0SE[i] + fTemp1SE[i];
    }
}
```

On voit cependant qu'il serait plus immédiat d'ajouter directement les résultats dans la sortie et économiser ainsi une boucle et de la place en mémoire :

```
void compute(int count, float** input, float** output) {
    for (int i = 0; i < count; i++) {
        output[0][i] = f1(input[0][i]);
    }

    for (int i = 0; i < count; i++) {
        output[0][i] += f2(input[0][i]);
    }
}
```

Une séparation habile des boucles permettrait ainsi de générer du code plus lisible par les compilateurs, dont l'empreinte mémoire est mieux localisée, tout en conservant les capacités d'optimisation propres au langage Faust. C'est donc cette approche qui nous semble la plus prometteuse pour la suite.

Chapitre 5

Conclusion et perspectives

La motivation initiale de ce stage était de comprendre comment il est possible que le même code C++ puisse ne pas s'exécuter de la même manière lorsqu'on l'agence différemment. La question était d'autant plus importante que les différences observées sont loin d'être anodines, ce qui laissait espérer des gains de performances significatifs si l'on trouvait une explication à ces différences.

L'intuition initiale était que les registres et le cache du processeur sont utilisés différemment lorsque l'ordre change. On a vu que si c'est effectivement le cas, d'autres mécanismes entrent en jeu : la génération de séquences d'instructions indépendantes les unes des autres semble favorable à de meilleures optimisations par les compilateurs, et à une meilleure utilisation du pipeline et du superscaling des processeurs.

En étoffant nos mesures et en s'appuyant largement sur les Performance Monitoring Units, nous avons finalement conçu un outil de mesure des performances du code Faust d'un nouveau genre, sobrement baptisé « Faust Scheduling Tools », dont le code est maintenant publié sur internet. Cet outil de mesure ne se contente pas de mesurer le débit, en échantillons par seconde, produit par un DSP, mais analyse dans le détail son impact sur les différents composants du processeur.

Cette capacité à mesurer de manière systématique le comportement détaillé d'un programme Faust dépasse le cadre de l'optimisation des techniques d'ordonnancement : toutes les options du compilateur Faust peuvent ainsi être testées sur un programme pour produire des observations bien plus fines que le seul temps d'exécution. On peut également s'en servir pour tester des modifications manuelles du code généré, comme on l'a fait pour tester la séparation des boucles, avant de se lancer dans le développement d'une nouvelle fonctionnalité.

Pour la suite du développement du compilateur, on a dans un premier temps cherché à améliorer l'ordonnancement existant avec un nouvel algorithme inspiré du List Scheduling utilisé pour la génération de code assembleur. La stratégie obtenue, qui semble légèrement plus performante, peut sûrement être encore raffinée. L'algorithme proposé est très souple puisque tout repose sur le processus itératif de sélection d'une nouvelle instruction. Le nôtre est encore très simple et ne repose que sur la détection de lignes à retard, il est certainement possible de l'améliorer.

Enfin, les derniers essais de séparation de boucles ont montré qu'une marge de progression énorme existe encore. On comprend maintenant que le code généré par Faust gagnerait à utiliser des boucles plus simples, dont les données peuvent tenir, sinon dans les registres du processeur, du moins dans le plus bas niveau de cache, et pour lesquelles un compilateur moderne est capable de trouver davantage d'optimisations.

Bibliographie

[1] Yann Orlarey, Dominique Fober et Stéphane Letz : Faust : an efficient functional approach to dsp programming. *In* Editions DELATOUR FRANCE, éditeur : *New Computational Paradigms for Computer Music*, pages 65–96. 2009.

[2] Yann Orlarey, Stéphane Letz, Dominique Fober et Romain Michon : A new intermediate representation for compiling and optimizing faust code. *International Faust Conference*, Décembre 2020.

[3] John L. Hennessy et David A. Patterson : *Computer Architecture, Fifth Edition : A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th édition, 2011.

[4] Denis Bakhvalov : *Performance Analysis and Tuning on Modern CPUs.* Publié indépendamment, 2023.

[5] Sanjay M Krishnamurthy : A brief survey of papers on scheduling for pipelined processors. *ACM SIGPLAN Notices*, 25(7):97–106, 1990.

[6] Keith D. Cooper et Linda Torczon : *Engineering a Compiler.* Morgan Kaufman, 2012.

[7] Keith D. Cooper, Philip J Schielke et Devika Subramanian : An experimental evaluation of list scheduling. *TR98*, 326, 1998.