

# Profilage de code sous Windows et Linux



par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 03/06/2008

Dernière mise à jour :

Le profilage du code est un outil indispensable pour optimiser de manière pertinente son code. Avant de passer à l'optimisation, il est nécessaire de savoir où le processeur passe la majeure partie de son temps, et c'est le rôle du profileur.

Chaque plateforme a son profileur privilégié, sous Linux, il s'agit principalement de **Valgrind** et de **VTune** dans une moindre mesure (gratuits tous les deux, sous conditions pour VTune), sous Windows il n'existe pas de profileur gratuit, mais les outils les plus connus sont **VTune** et **Visual Studio** qui en dispose dans sa version **Team Suite**. VTune est le profileur d'Intel, **CodeAnalyst** est l'équivalent chez AMD. Ici ne seront présentés que Valgrind et VTune, en environnement multithread.

Le profil sera étudié avec un ray tracer interactif proposé dans **ce tutoriel**. Il s'agit d'une bibliothèque appelée depuis Python, donc la situation la plus complexe à profiler, naturellement compilée en mode optimisé.

I - Qu'est un profil de code ?.....	3
I-A - Profil par échantillonnage.....	3
I-B - Profil par instrumentation.....	3
I-C - Profil par émulation.....	3
II - Visual Studio Performance Tool.....	3
II-A - Mise en place.....	3
II-B - Echantillonnage.....	6
II-C - Instrumentation.....	9
III - Valgrind et Callgrind.....	11
III-A - Mise en place.....	11
III-B - Analyse.....	11
IV - Conclusion.....	14

## I - Qu'est un profil de code ?

Un profil de code est un outil indispensable permettant de repérer une fonction ou une instruction qui prennent trop de temps et qui doivent donc être refactorisées. Le repérage de ces fonctions ou instructions est un exercice difficile et en général mal fait si un outil de profilage n'est pas utilisé.

Un dicton indique que 80% du temps est passé dans 20% du code (on entend parfois aussi ce dicton avec les chiffres 90 et 10), ce sont ces 20% qui doivent être repérés.

Plusieurs méthodes de profilage existent. Chaque méthode a ses avantages et ses inconvénients. Il est aussi à noter que le temps d'exécution ne fait pas tout, le nombre d'instructions, absolu et par seconde, leur type, le comportement du cache, ... tous ces éléments peuvent être mesurés et donnent des informations.

### I-A - Profil par échantillonnage

Le profil par échantillonnage consiste à regarder tous les  $n$  cycles/instructions/défauts de cache/... quelle est la fonction en cours d'exécution, ainsi que les fonctions parentes.

Ce type de profil est très facile à réaliser, le profileur exécutant simplement le code et mesurant régulièrement des valeurs : pas d'intrusion, vitesse quasi-identique à la réalité. En revanche, le profil n'est pas exact et donc doit être pris avec précaution. Il est nécessaire de tester avec plusieurs valeurs de fréquence d'échantillonnage.

### I-B - Profil par instrumentation

L'instrumentation modifie le programme testé en ajoutant à chaque appel de fonction une fonction d'instrumentation. Celle-ci va mesurer le temps d'appel, le nombre d'instructions exécutés, ... Tout cela grâce à des compteurs internes au processeur qui permet d'accéder à ces valeurs.

Ce profil est intermédiaire entre l'échantillonnage et l'émulation. Il est plus intrusif que les deux autres, le programme étant modifié, mais le résultat est plus exact. En revanche, du code supplémentaire est exécuté, donc le résultat n'est pas exact.

### I-C - Profil par émulation

L'émulation consiste à exécuter le programme sur un processeur virtuel. Tout peut être mesuré de manière exacte.

Le principal inconvénient de cette méthode est sa lenteur. En effet, l'émulation d'un processeur est complexe. De plus, le processeur émulé n'est pas le processeur utilisé, donc les résultats ne sont pas généralisables. En revanche, pour le processeur émulé, ils sont exacts.

## II - Visual Studio Performance Tool

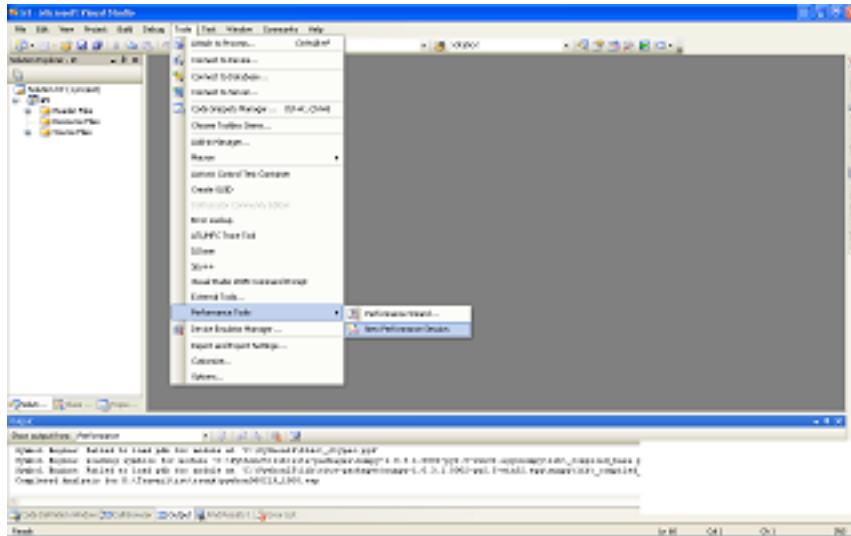
Visual Studio propose dans sa version **Team Suite** un profileur par échantillonnage et par instrumentation. Il est complètement intégré dans l'IDE et peut être géré par une solution Visual Studio.

Deux types de profil peuvent être générés, par échantillonnage ou par instrumentation.

### II-A - Mise en place

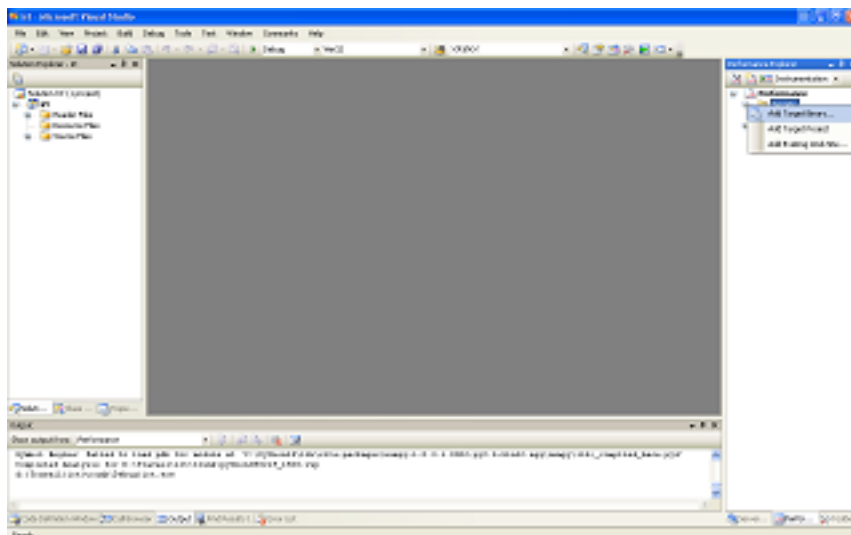
Tout d'abord, le code doit être compilé avec l'option **/Zi** ou **/Z7**. Ensuite, l'édition des liens doit s'effectuer avec **/DEBUG**. Sans cela, il sera impossible de mesurer quelque chose.

Dans mon cas précis, je n'ai pas de projet Visual Studio, donc je commence par créer une solution avec mes fichiers sources pour pouvoir les parcourir en parallèle du profil.



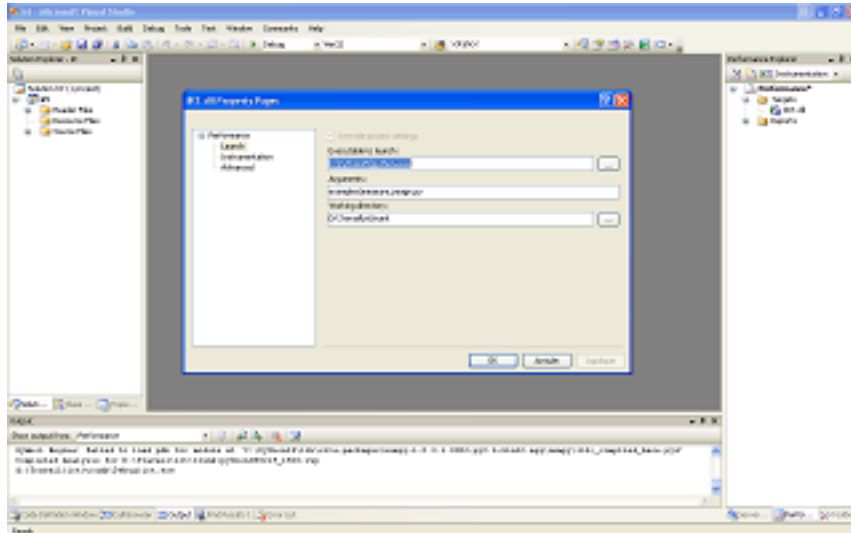
Lancement du profileur

Maintenant, il est nécessaire d'indiquer quels seront les cibles à profiler.

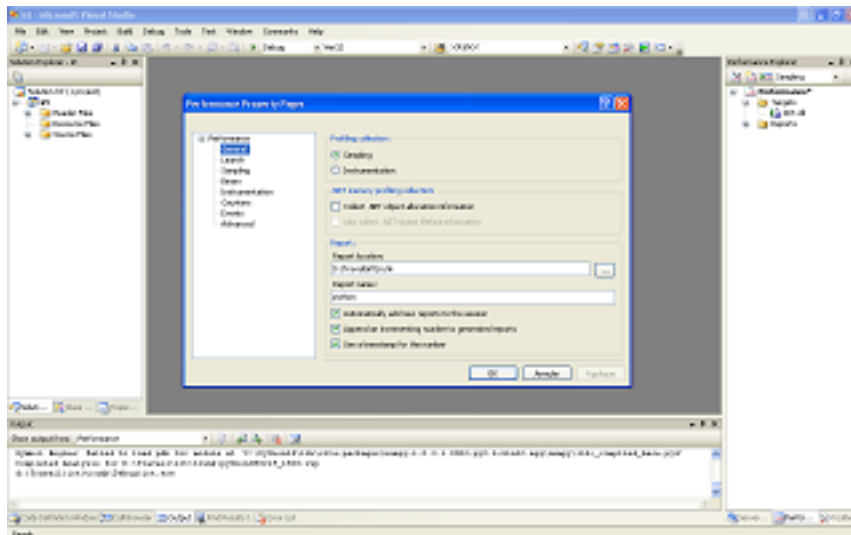


Ajouter un binaire

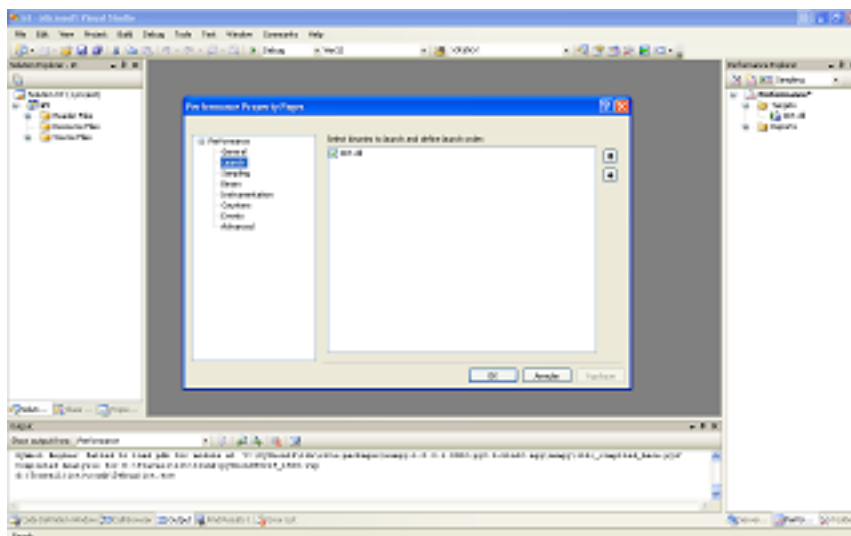
Les propriétés de la cible peuvent être modifiées, en particulier la manière d'exécuter le code. Ici, il s'agit d'un script Python exécuté dans le dossier du raytracer.



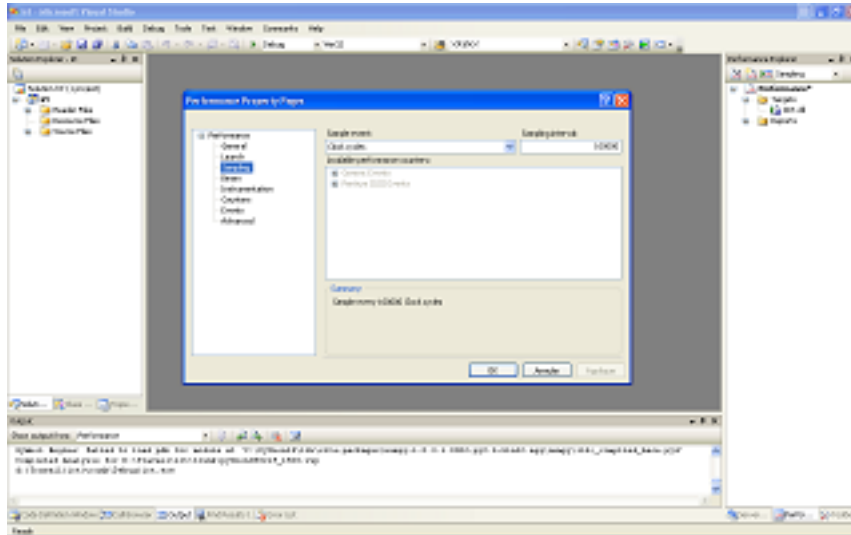
Propriétés du binaire



Propriétés générales du profil



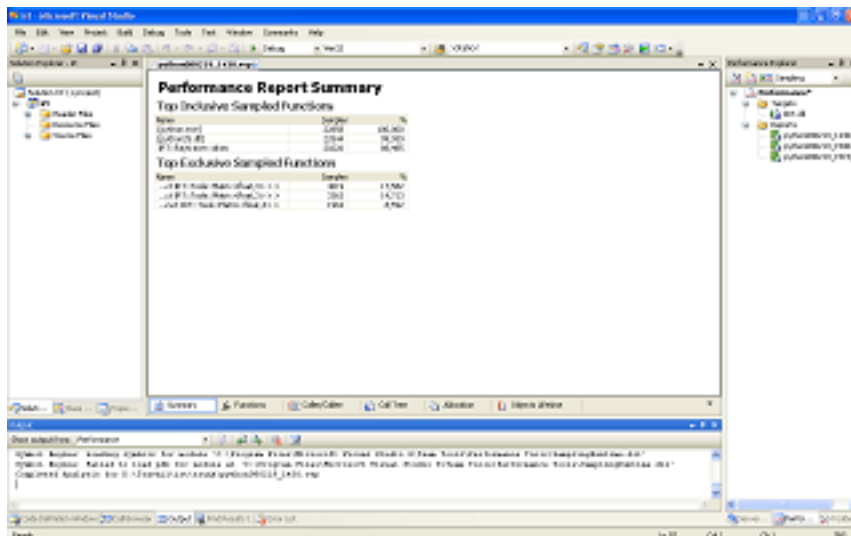
Ajout des cibles à profiler



Echantillonnage sur évènements

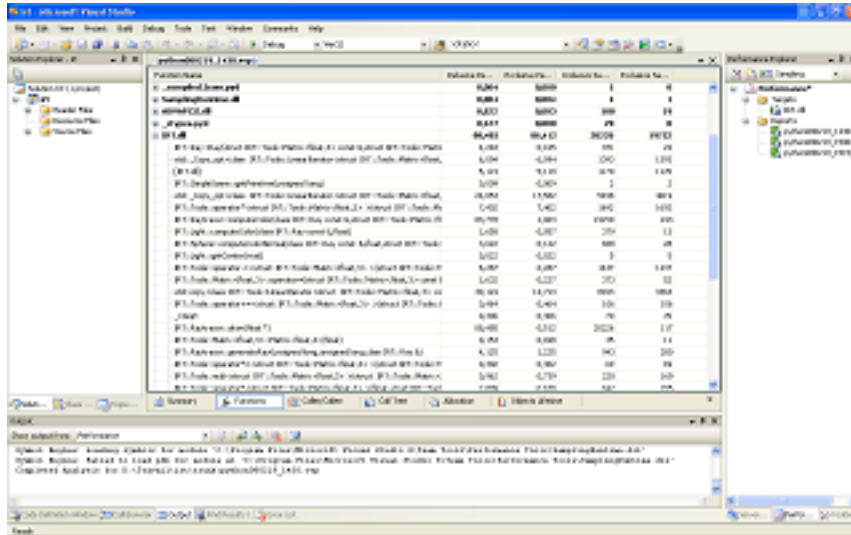
## II-B - Echantillonnage

Une fois le profil exécuté, la page principale reprend les fonctions dans lesquelles le maximum de temps a été passé, de manière inclusive ou exclusive. Inclusive signifie que c'est la fonction ou les fonctions appelées qui seront mesurées, exclusive ne donne que la fonction mesurée.



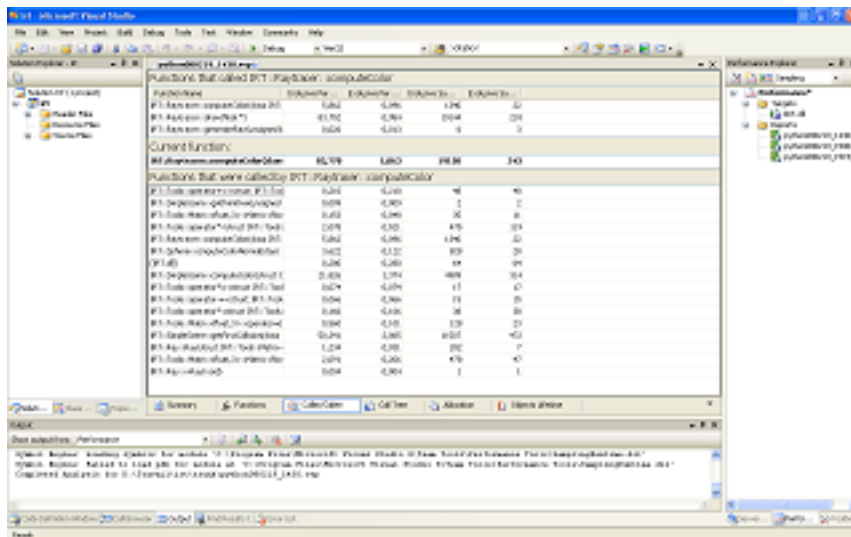
Page récapitulative du profil

La liste des fonctions est organisée selon les bibliothèques et exécutables. Dans ce tableau sont donnés le temps passé absolu inclusif ou exclusif donné en nombre d'échantillons, mais aussi en pourcentage.



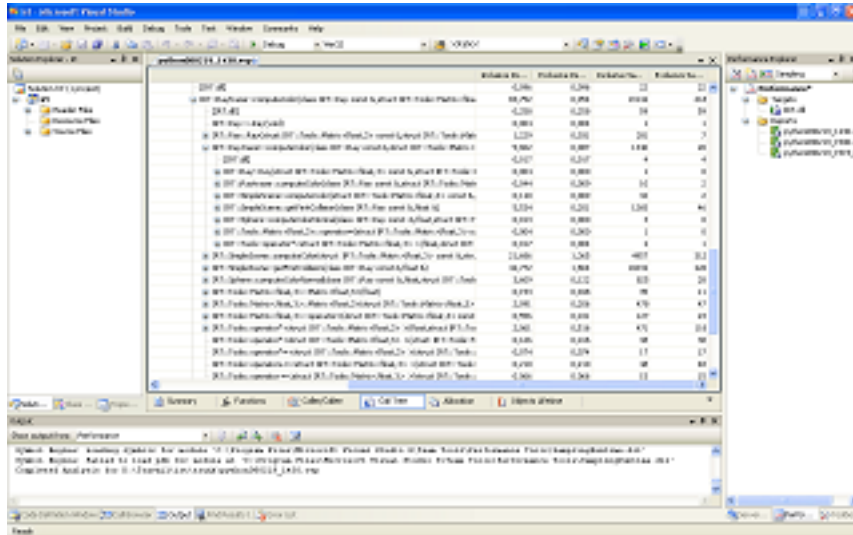
Liste des fonctions échantillonnées

Une fois une fonction sélectionnée, les fonctions appelées et appelantes peuvent être affichées et analysées pour trouver un point chaud commun.



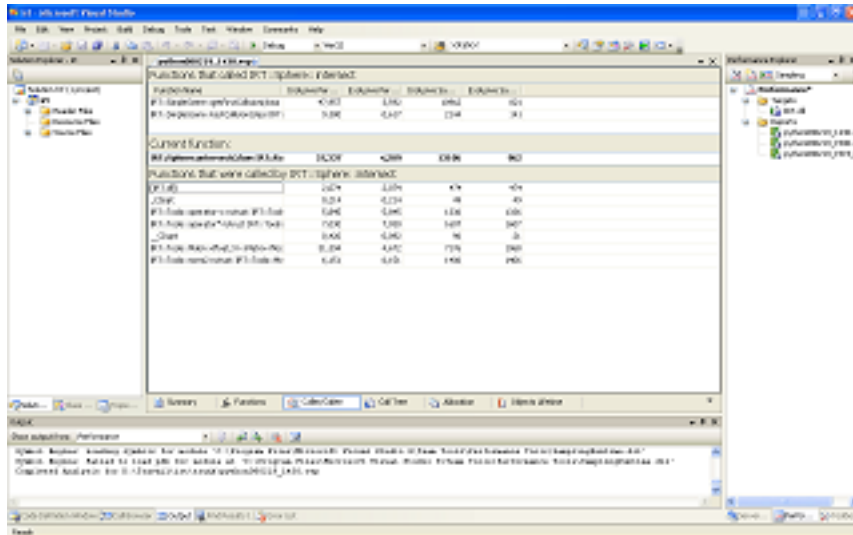
Fonctions appelantes et appelées d'une fonction précise

Un arbre des appels est aussi fourni, mais il n'est pas très utilisable, malheureusement.



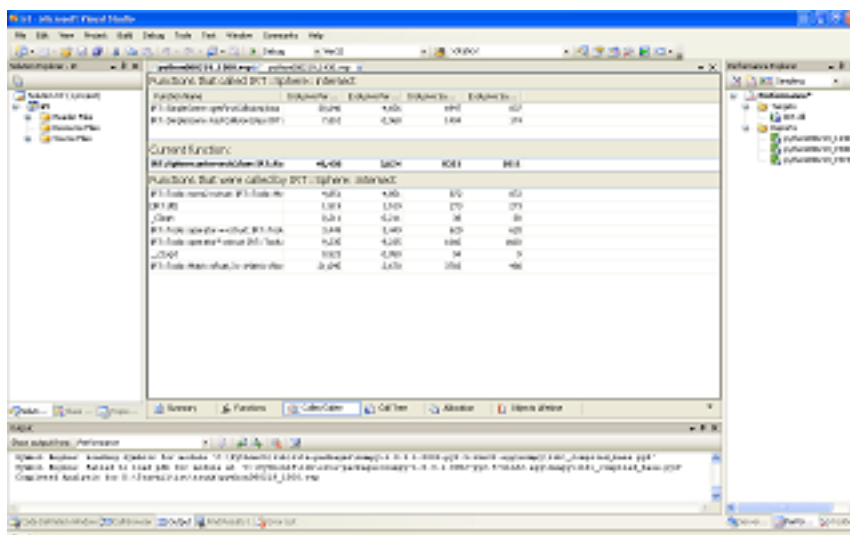
Arbre des appels

Maintenant, voyons comment utiliser les résultats du profileur. La fonction la plus critique va être affichée et analysée.




La fonction la plus critique du programme

La sous-fonction la plus critique dans cette fonction occupe 31% du programme total, il s'agit d'un constructeur d'un objet. L'optimisation envisagée va réduire le nombre d'appels à ce constructeur et le pourcentage d'utilisation passe à 21%. Ce gain sur cette fonction est important et se traduit sur l'ensemble du programme par un gain de près de 30%. Le profil ici est donc incontournable.



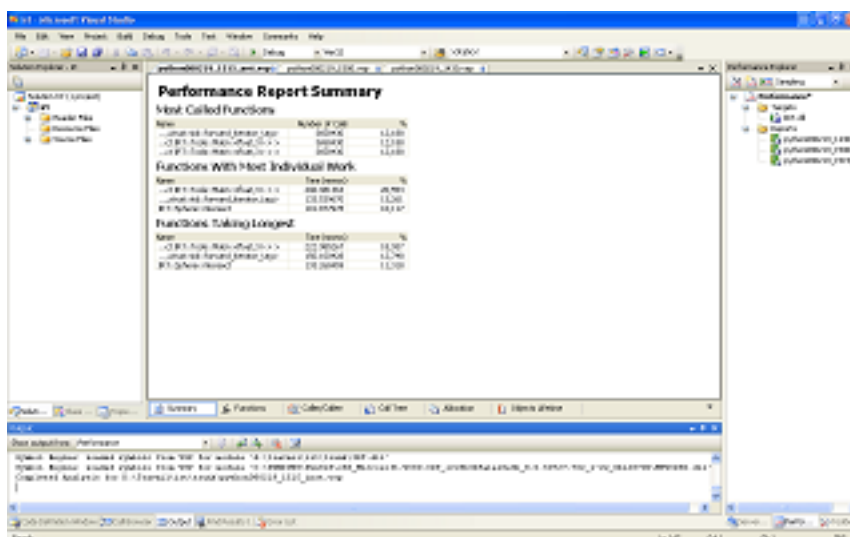
La fonction la plus critique après optimisation du programme

 Avec le bouton droit, le menu contextuel permet d'accéder directement à la fonction étudiée (pratique).

## II-C - Instrumentation

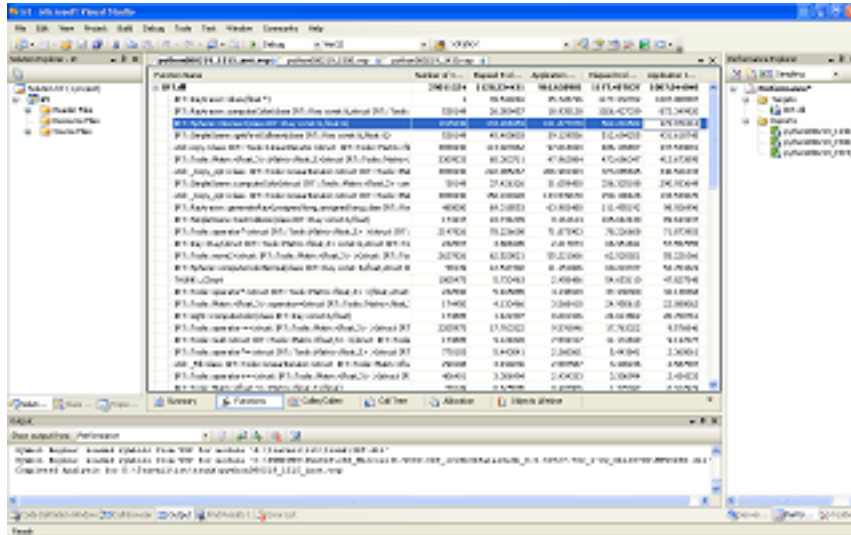
La sélection de l'instrumentation est facile : il suffit de choisir l'instrumentation à la place de l'échantillonnage et de relancer un profil.

La page principale de ce type de profil est différente : elle récapitule les fonctions les plus appelées, celles qui ont un coût inclusif important (s'il s'agit d'une fonction très appelée et très longue, elle est à optimiser) et celles qui ont un coût inclusif élevé.



Page récapitulative du profil

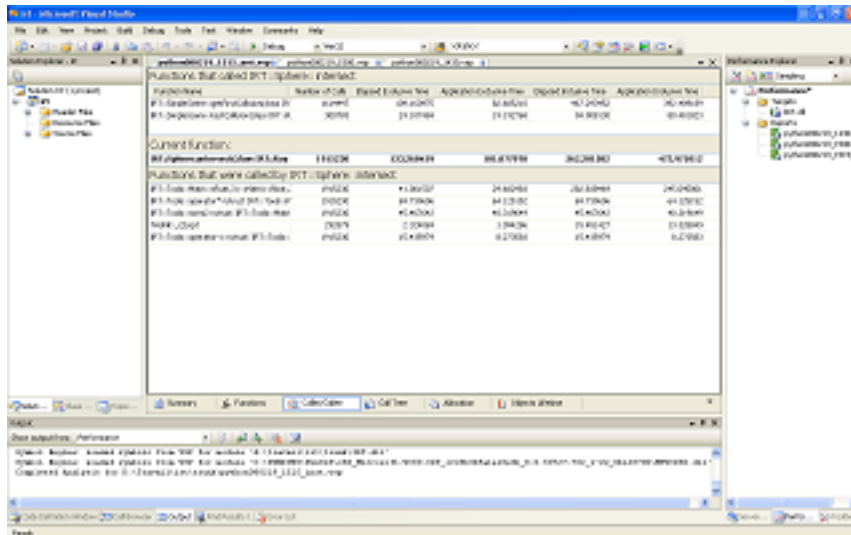
La liste des fonctions instrumentées indique le nombre d'appels, le temps passé inclusif et exclusif, et enfin le temps réel passé dans l'application.



Nom	Nombre d'appels	Temps écoulé	Application	Temps d'application
main	1	0:00:00.000	...	0:00:00.000
...	...	...	...	...

Liste des fonctions instrumentées

Ici, le tableau est identique à l'échantillonnage, mais avec les temps calculés par l'instrumentation. De même pour l'arbre des appels.

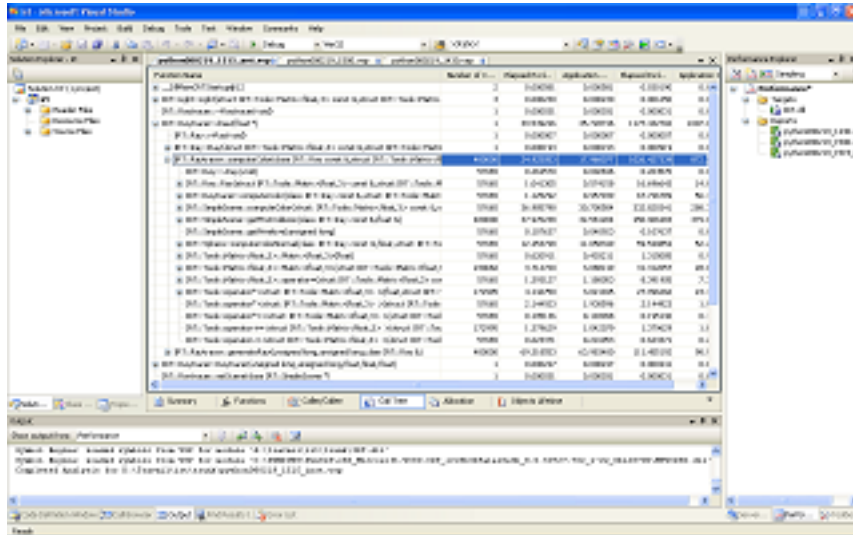


Parent Functions	Number of Calls	Elapsed Time	Application	Application Time
main	1	0:00:00.000	...	0:00:00.000

Current Function	Number of Calls	Elapsed Time	Application	Application Time
main	1	0:00:00.000	...	0:00:00.000

Fonctions appelantes et appelées d'une fonction précise



Arbre des appels

### III - Valgrind et Callgrind

Valgrind est en réalité bien plus qu'un profileur, c'est un émulateur de processeur. Greffée par dessus se trouve une liste d'outils :

- memcheck, qui vérifie les fuites mémoires, les dépassements,
- cachegrind, qui mesure des données par rapport au cache,
- callgrind, qui est le profileur en question.

Valgrind génère un fichier de résultat qui pourra être analysé par **KCacheGrind**.

💡 Pour ajouter les informations de débogage avec gcc, il suffit de compiler avec l'option **-g**.

#### III-A - Mise en place

Valgrind se lance en ligne de code à l'aide de cette commande :

```
valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --compute-jumps=yes python exemples/measure_image.py
```

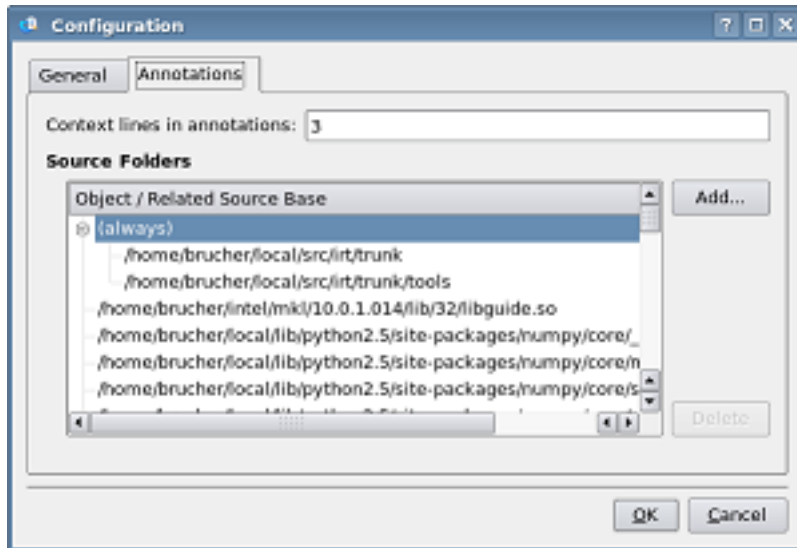
**--dump-instr=yes** permet d'enregistrer les instructions exécutées, indispensable pour comparer avec le code source, **--simulate-cache=yes** ajoute les coûts liés au cache, **--compute-jumps=yes** ajoute les sauts dans le rapport.

💡 Contrairement au profileur de Visual Studio, à part les instructions et le cache, les autres coûts ne peuvent être mesurés (comme le nombre d'instructions flottantes, vectorielles, ...).

#### III-B - Analyse

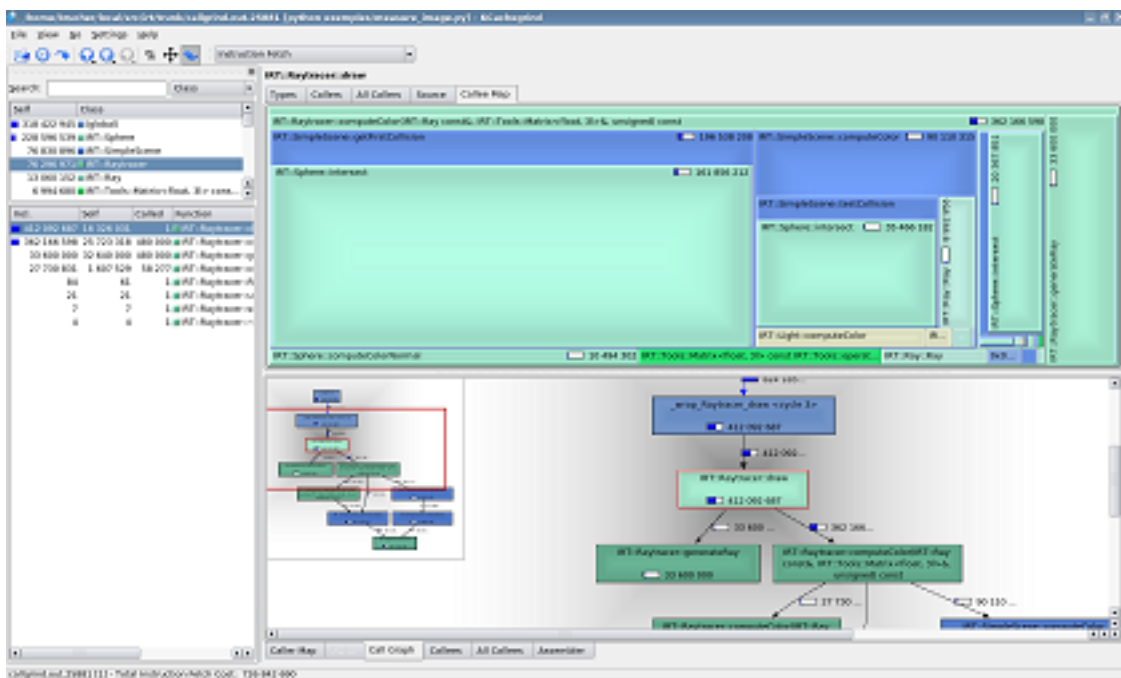
Une fois le profil enregistré (un fichier **callgrind.out.n**), il reste à l'analyser.

Afin de pouvoir analyser le code source, il est nécessaire de configurer KCacheGrind (menu Options) et d'ajouter les dossiers où se trouvent les fichiers sources.

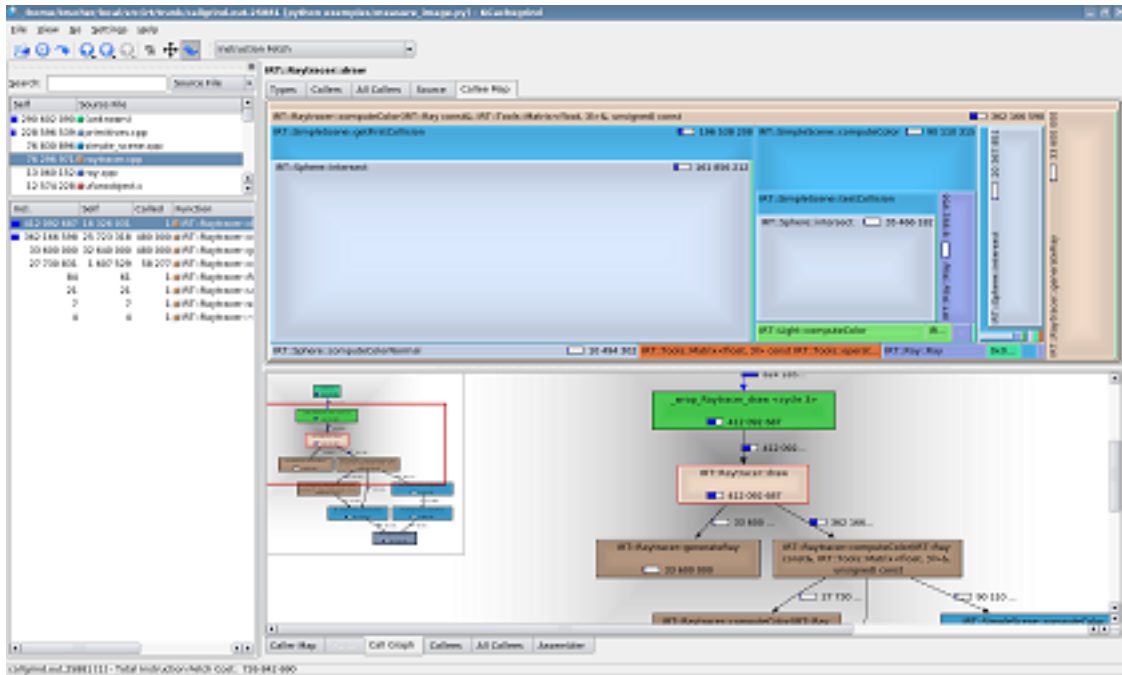


*Carte et graphe des appels triés par classe*

Le rapport peut être classé par classe, par fichier source, ... La différence se situe au niveau des couleurs et au niveau des informations données en parallèle. Dans le cas suivant, le classement par classe ou par fichier source est très semblable (comme il y a un fichier source par classe...).



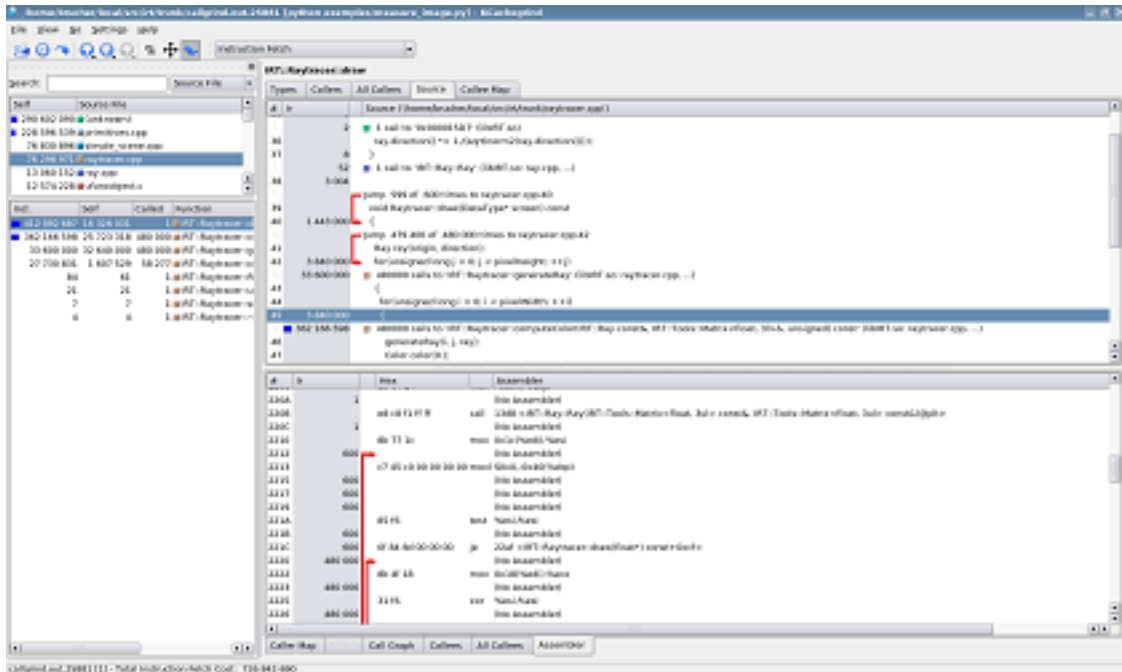
*Carte et graphe des appels triés par classe*



Carte et graphe des appels triés par fichier source

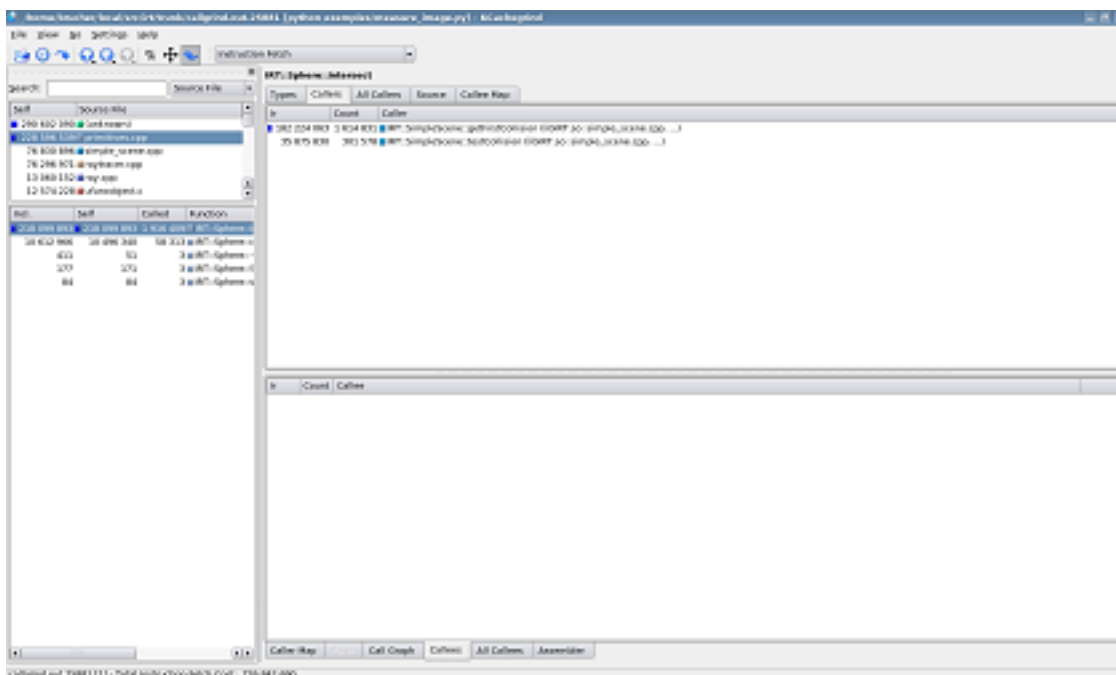
Une fois une fonction sélectionnée, le code (source et assembleur) peut être affiché.

**⚠** En mode optimisé, gcc ne fournit pas toutes les informations du code source. Les fonctions ne peuvent pas toutes être analysées (comme dans le débogueur gdb), même si KCacheGrind affiche le maximum de ce qu'il peut faire. Il s'agit d'un inconvénient majeur, à mon avis, de gcc sur Visual Studio.



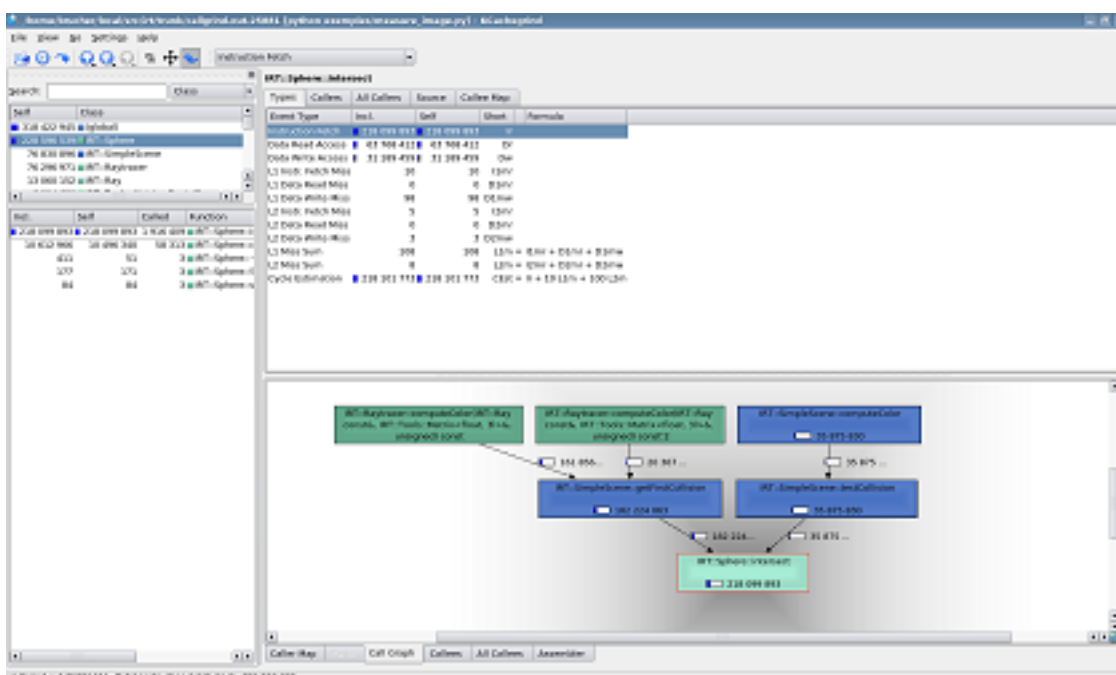
Coûts proposés

Tout comme Visual Studio, il est possible d'afficher les fonctions appelantes et appelées.



Coûts proposés

Enfin, tous les coûts sont affichables pour une même fonction.



Coûts proposés

## IV - Conclusion

Chaque profileur a ses particularités, ses forces et ses faiblesses. L'outil de Visual Studio est performant, dommage qu'il ne soit pas disponible dans toutes les versions, ou au moins dans les versions professionnelles, car il permet de récupérer un grand nombre d'informations.

Valgrind est un outil qui est gratuit, mais très puissant. Même s'il ne fait qu'émuler un processeur (et donc l'exécution d'un programme est lente, même s'il est possible de spécifier des morceaux de code à profiler, cela n'a pas été le

cas ici, il s'agit d'une introduction/présentation), l'alliance avec KCacheGrind est explosive, ce dernier étant bien plus agréable à manipuler que Visual Studio.

Je n'ai pas testé tous les outils existants, mais Visual Studio et Valgrind sont agréables à utiliser et ne m'ont jamais fait défaut (contrairement à d'autres profileurs très connus).