# Manuel de KCachegrind

**auteur original de la documentation: Josef Weidendorfer**
**Mises à jour corrections: Federico Zenith**
**Traduction française : Yann Verley**
**Traduction française : Ludovic Grossard**
**Traduction française : Damien Raude-Morvan**

# Table des matières

**Résumé**

KCachegrind est un outil d'affichage de données de profilage, écrit en utilisant l'environnement KDE Frameworks 5.

# Chapitre 1

# Introduction

KCachegrind est un navigateur pour les données produites par des outils de profilage. Ce chapitre explique à quoi sert le profilage, comment cela fonctionne et donne quelques exemples d'outils de profilage disponibles.

## 1.1 Profilage

Habituellement, quand on développe un programme, une des dernières étapes est d'optimiser les performances. C'est une perte de temps d'optimiser les fonctions rarement utilisées. Il est donc intéressant de savoir où votre programme passe le plus de temps.

Pour du code séquentiel, la réception des données statistiques des caractéristiques de l'exécution des programmes, comme par exemple le temps passé dans les fonctions ou dans les lignes de code est habituellement suffisant. C'est ce que l'on appelle Profiler. Le programme est exécuté sous le contrôle d'un outil de profilage, qui donne les résultats de l'exécution à la fin. Au contraire, pour du code parallèle, les problèmes de performance proviennent généralement de l'attente par un processeur de données d'un autre processeur. Comme ce temps d'attente ne peut habituellement être facilement attribué, il est préférable de générer ici des traces d'évènements horodatées. KCachegrind ne peut pas afficher ce type de données.

Après analyse des données de profilage produites, il devrait être plus facile de voir les points chauds et les goulots d'étranglement du code. Par exemple, on peut vérifier les hypothèses par rapport au nombre d'appels, et les régions identifiées du code peuvent être optimisées. Après cela, on doit valider l'optimisation effectuée avec une autre exécution profilée.

## 1.2 Méthodes de profilage

La mesure exacte du temps passé par des évènements se produisant pendant l'exécution d'une région de code (par exemple, une fonction) nécessite que soit effectué un ajout de code de mesure avant et après cette région. Ce code lit le temps ou bien un compteur global d'évènement, et calcule les différences. Le code original doit ainsi être changé avant l'exécution. C'est ce que l'on appelle l'instrumentation. L'instrumentation peut être faite par le programmeur lui-même, le compilateur, ou bien par le système d'exécution. Comme les régions intéressantes sont généralement imbriquées, la surcharge due à la mesure influence toujours la mesure elle-même. Ainsi, l'instrumentation doit être effectuée sélectivement et les résultats doivent être interprétés avec précaution. Bien sûr, ceci fait que l'analyse des performances se basant sur la mesure exacte est un processus très complexe.

La mesure exacte est possible grâce à des compteurs matériels (ce qui inclut des compteurs s'incrémentant sur un tic de temps) fournis dans les processeurs modernes et qui sont incrémentés

quand un évènement se produit. Comme nous voulons attribuer des évènements à des régions de code, sans utiliser de compteurs, nous devons gérer chaque évènement en incrémentant un compteur pour la région de code courante nous-mêmes. Faire ceci au niveau logiciel n'est bien sûr pas possible. Toutefois, si on part de l'hypothèse que la distribution des évènements sur le code source est identique quand on regarde à chaque énième évènement, au lieu de chaque évènement, nous avons construit une méthode de mesure réglable par rapport à la surcharge induite. C'est ce que l'on appelle l'échantillonnage. L'échantillonnage à base de temps (NdT : *Time Based Sampling* ou TBS) utilise un temporisateur pour regarder régulièrement le compteur de programme, afin de créer un histogramme sur le code du programme. L'échantillonnage à base d'évènements (NdT : *Event Based Sampling* ou EBS) se sert des compteurs matériels des processeurs modernes et utilise un mode dans lequel le gestionnaire d'interruptions est appelé sur les valeurs basses du compteur, en générant un histogramme de la distribution d'évènements correspondante. Dans le gestionnaire, le compteur d'évènement est toujours réinitialisé au  n de la méthode d'échantillonnage. L'avantage de l'échantillonnage est que le code n'a pas besoin d'être modifié, mais ceci reste un compromis : la supposition d'au-dessus est correcte si n est petit, mais plus n est petit, plus la surcharge du gestionnaire d'interruptions est importante.

Il existe une autre méthode de mesure qui est de simuler ce qui arrive au niveau de l'ordinateur quand on exécute un code donné, c'est-à-dire une simulation contrôlée de code. La simulation est toujours dépendante du modèle de la machine qui est plus ou moins précis. Cependant, pour des modèles très détaillés de machine, s'approchant de la réalité, le temps de simulation peut être assez inacceptable pour une utilisation courante. L'avantage est que l'on peut insérer dans un code donné un code de mesure / simulation aussi complexe qu'il soit sans perturber les résultats. Faire ceci directement avant l'exécution (ce que l'on appelle instrumentation dynamique), en utilisant le binaire original, est très confortable pour l'utilisateur : aucune recompilation n'est nécessaire. Cette méthode devient utilisable quand on ne simule que quelques parties de la machine avec un modèle simple. En outre, les résultats produits par des modèles simples sont souvent plus faciles à comprendre : le problème fréquent avec le vrai matériel est que les résultats incluent des effets de chevauchement de différentes parties de la machine.

## 1.3 Outils de profilage

Le plus connu des outils de profilage est l'outil de la suite GCC, gprof : on doit compiler le programme avec l'option `-pg` ; le lancement du programme génère un fichier `gmon.out`, que l'utilisateur peut lire avec **gprof**. L'inconvénient principal de cette méthode est l'obligation de passer par une recompilation pour préparer l'exécutable, qui doit être lié de façon statique. La méthode utilisée ici est l'instrumentation générée par le compilateur. Celle-ci mesure les arcs d'appels se produisant dans les fonctions et en accord avec des compteurs d'appels, en conjonction avec un TBS, qui donne un histogramme de distribution du temps sur le code. En utilisant les deux informations, il est possible de calculer de manière heuristique le temps d'inclusion des fonctions, c'est-à-dire le temps passé dans une fonction ainsi que toutes les fonctions qu'elle a appelées.

Pour une mesure exacte des évènements, il existe des librairies avec des fonctions capables de lire les compteurs de performance matériels. Les plus connus sont le patch PerfCtr pour Linux® et les librairies indépendantes de l'architecture PAPI et PCL. Comme toujours, une mesure exacte nécessite une instrumentation du code, comme indiqué au-dessus. D'autres utilisent les librairies elles-mêmes ou utilisent des systèmes d'instrumentation automatiques comme ADAPTOR (pour l'instrumentation de sources FORTRAN), ou encore DynaProf (injection de code par DynInst).

OProfile est un outil de profilage au niveau système pour Linux® utilisant l'échantillonnage.

Dans beaucoup d'aspects, une manière agréable de profiler est d'utiliser Cachegrind ou Callgrind, qui sont des simulateurs utilisant l'environnement d'instrumentation d'exécution Valgrind. Comme il n'y a pas besoin d'accéder aux compteurs hardware (souvent difficile avec les installations Linux® actuelles), et comme les binaires devant être profilés n'ont pas besoin d'être modifiés, ceci est une bonne alternative à d'autres outils de profilage. L'inconvénient du ralentissement dû à la simulation peut être réduit en n'effectuant la simulation que sur les parties intéressantes du programme, et peut-être seulement sur quelques itérations d'une boucle. Sans

instrumentation de la mesure/simulation, l'usage de Valgrind ne provoque qu'un ralentissement d'un facteur de 3 à 5. Et si on n'est intéressé que par l'arbre d'appels et le nombre d'appels, le simulateur du cache peut être désactivé.

La simulation du cache est la première étape dans l'approximation des temps réels. En effet, sur les systèmes modernes, l'exécution est très sensible à l'exploitation de ce qu'on appelle des *caches* (zones de mémoire petites et rapides, et qui permettent d'accélérer les accès répétés aux mêmes emplacements mémoire). Cachegrind fait cette simulation du cache en interceptant les accès mémoires. Les données produites incluent le nombre d'accès sur la mémoire des instructions / des données, les échecs des caches de niveau L1 / L2, et elle met en relation les lignes du code source avec les fonctions du programme exécuté. En combinant ces compteurs d'échecs et en utilisant des temps de latence de processeurs connus, on peut faire une estimation du temps passé.

Callgrind est une extension de Cachegrind qui construit l'arbre d'appels d'un programme à la volée, c'est-à-dire comment les fonctions s'appellent entre elles et combien d'évènements se produisent lors de l'exécution d'une fonction. De plus, les données de profilage devant être collectées peuvent être divisées en threads ou en contextes de chaînes d'appels. Il peut aussi fournir des données de profilage au niveau instruction afin de permettre l'annotation d'un code désassemblé.

## 1.4 Affichage

Les outils de profilage produisent typiquement un nombre important de données. Le souhait de naviguer facilement dans l'arbre d'appels, ainsi que de passer rapidement d'un mode de tri des fonctions et d'affichage des différents types d'évènements, a motivé la création d'une interface graphique (GUI) pour accomplir cela.

cette application est un outil d'affichage de données de profilage permettant d'accomplir ces souhaits. Il a été programmé en premier lieu pour naviguer dans les données de Cachegrind et Calltree. Cependant, il existe des convertisseurs permettant d'afficher les données de profilage produites par d'autres outils. Une description du format des fichiers Cachegrind / Callgrindest donnée dans l'annexe.

En plus d'une liste de fonctions triées en fonction des métriques des coûts inclusifs ou exclusifs et de manière optionnelle groupées par fichier source, librairie partagée ou classe C++, KCachegrind propose des affichages différents pour une fonction sélectionnée, à savoir :

— une vue de l'arbre d'appel, qui montre une section de l'arbre d'appel autour de la fonction sélectionnée,

— une vue de la carte de l'arbre, qui permet d'afficher la relation entre appels imbriqués ainsi que la métrique du coût inclusif pour détecter visuellement et rapidement les fonctions à problèmes,

— les vues du code source et de l'annotation assembleur, permettant de voir les détails des coûts associés aux lignes du code source et des instructions assembleur.

# Chapitre 2

# Utiliser KCachegrind

## 2.1 Générer les données à afficher

Tout d'abord, il faut générer les données de performance en mesurant les aspects des caractéristiques de l'exécution d'une application. Pour cela, il faut utiliser un outil de profilage. KCachegrind n'inclut pas d'outil de profilage, mais est prévu pour fonctionner avec Callgrind. En utilisant un convertisseur, il peut aussi être utilisé pour afficher les données produites par OProfile. Même si l'objectif de ce manuel n'est pas de documenter le profilage avec ces outils, la prochaine section fournit des petits tutoriels afin que vous puissiez démarrer.

### 2.1.1 Callgrind

Callgrind est disponible sur Valgrind. Notez qu'il était précédemment nommé Calltree, mais ce nom était trompeur.

L'usage le plus répandu est de démarrer votre application en préfixant la ligne de commande par **`valgrind --tool=callgrind`** , comme dans :

```
valgrind --tool=callgrind mon_programme mes_arguments
```

À la fin de l'exécution du programme, un fichier `callgrind.out.pid` sera généré. il peut être chargé dans KCachegrind.

Un usage plus avancé est de générer des données de profilage quand une fonction donnée de votre application est appelée. Par exemple, pour Konqueror, pour n'avoir les données de profilage que pour le rendu d'une page web, vous pouvez décider de générer les données quand vous sélectionnez l'élément du menu **Affichage** → **Recharger** . Ceci correspond à un appel à `KonqMainWindow::slotReload`. Utilisez :

```
valgrind --tool=callgrind --dump-before=KonqMainWindow::slotReload konqueror
```

Ceci va produire plusieurs fichiers de données de profilage avec un numéro additionnel séquentiel à la fin du nom du fichier. Un fichier sans un tel nombre à la fin (se terminant seulement par le PID du processus) sera aussi produit. En chargeant ce fichier dans KCachegrind, tous les autres fichiers seront aussi chargés, et peuvent être affichés dans la **synthèse des parties** et dans la **liste des parties**.

### 2.1.2 OProfile

OProfile est disponible sur sa page web. Suivez les instructions d'installation du site web. Veuillez vérifier toutefois si votre distribution ne le fournit pas déjà en tant que paquet (comme dans SuSE®).

Le profilage au niveau système n'est autorisé que pour l'administrateur, car toutes les actions sur le système peuvent être observées. C'est pourquoi ce qui va suivre doit être fait en tant qu'administrateur. Tout d'abord, configurez le processus de profilage, en utilisant l'interface graphique avec **oprof_start** ou l'outil en ligne de commande **opcontrol**. Une configuration standard devrait être le mode temps (TBS, voir introduction). Pour démarrer la mesure, lancez **`opcontrol -s`**. Ensuite lancez l'application à profiler, et après, lancez **`opcontrol -d`**. Ceci va écrire les résultats de la mesure dans des fichiers sous le dossier `/var/lib/oprofile/samples/` . Pour pouvoir afficher les données dans KCachegrind, lancez la commande suivante dans un dossier vide :

```
opreport -gdf | op2callgrind
```

. Ceci va produire un nombre important de fichiers, un pour chaque programme qui s'exécutait sur le système. Chacun peut être chargé indépendamment dans KCachegrind.

## 2.2 Bases de l'interface utilisateur

Quand vous lancez KCachegrind avec un fichier de données de profilage en tant qu'argument, ou après en avoir chargé un avec **Fichier** → **Ouvrir**, Vous verrez une barre sur le côté contenant la liste des fonctions à gauche et, à droite de la partie principale, une aire d'affichage pour la fonction sélectionnée. L'aire d'affichage peut être configurée pour afficher plusieurs vues dans une seule.

Lors du premier lancement, cette zone sera divisée horizontalement en deux parties, une supérieure et une inférieure, toutes deux dotées de vues sélectionnables par onglets. Pour déplacer une vue, utilisez le menu contextuel des onglets, et ajustez les séparations entre vues. Pour passer rapidement d'une présentation de vue à une autre, utilisez **Vue** → **Disposition des vues** → **Aller à la disposition suivante** (**Ctrl+**→) et **Vue** → **Disposition des vues** → **Aller à la disposition précédente** (**Ctrl+**←)

Le type d'évènement actif est important pour l'affichage : pour Callgrind, c'est par exemple le nombre d'échecs du cache ou l'estimation du cycle ; pour OProfile, c'est le "temps" dans le plus simple cas. Vous pouvez changer le type d'évènement par une combobox dans la barre d'outils ou dans la vue du**type d'évènement**. Un premier aperçu des caractéristiques de l'exécution devrait être donné quand vous sélectionnez la fonction `main` dans la liste de gauche, et regardez l'affichage de l'arbre d'appels. Là vous voyez les appels se produisant dans votre programme. Notez que la vue du graphe d'appels ne montre que les fonctions avec un nombre d'évènements élevé. En double-cliquant sur une fonction dans le graphe, celui-ci change pour afficher les fonctions appelées autour de celle sélectionnée.

Pour explorer plus profondément GUI, vous pouvez regarder, en plus de ce manuel, la section de documentation du site Internet. De plus, chaque composant graphique de KCachegrind est fourni avec l'aide "Qu'est-ce que c'est ? ".

# Chapitre 3

# Concepts de base

Ce chapitre explique quelques concepts de KCachegrind, et introduit les termes utilisés dans l'interface.

## 3.1 Le modèle de données pour les données de profilage

### 3.1.1 Entités de coût

Les compteurs de coût des types d'évènement (comme les échecs du cache L2) sont attribués aux entités de coût, qui sont des éléments en relation avec le code source ou des structures de données d'un programme donné. Les entités de coût ne sont pas seulement un code simple ou des positions de données, mais aussi des tuples de position. Par exemple, un appel a une source et une cible, ou bien une adresse de données peut avoir un type de données et une position dans le code où son allocation s'est effectuée.

Les entités de coût connues de KCachegrind sont données ci-dessous. Les positions simples :

**Instruction**
: Une instruction assembleur à l'adresse spécifiée.

**Ligne dans le source d'une fonction**
: . Toutes les instructions que le compilateur (par l'intermédiaire des informations de débogage) associe à une ligne donnée spécifiée par le nom du fichier source et le numéro de la ligne, et qui sont exécutées dans le contexte de quelques fonctions. Le dernier est nécessaire parce qu'une ligne source à l'intérieur d'une fonction inline peut apparaître dans le contexte de fonctions multiples. Les instructions sans association avec une ligne du code source courant sont associées à la ligne numéro 0 du fichier « ??? ».`???`.

**Function**
: Fonction. Toutes les lignes d'une fonction donnée constituent la fonction elle-même. Une fonction est spécifiée, s'ils sont disponibles, par son nom et sa position dans quelques objets binaires. La dernière est nécessaire parce que les objets binaires d'un seul programme peut avoir des fonctions avec le même nom (on peut y accéder par exemple avec `dlopen` ou `dlsym`; l'éditeur de lien dynamique résout les fonctions dans un ordre de recherche donné dans les objets binaires utilisés). Si un outil de profilage ne peut détecter le nom du symbole d'une fonction, par exemple parce que l'information de débogage n'est pas disponible, soit l'adresse de la première instruction exécutée est utilisée, soit `???`

**Objet binaire.**
: Toutes les fonctions dont le code se situe à l'intérieur d'un objet binaire, ou bien l'exécutable principal (NdT : « main »), ou encore une librairie partagée.

**Fichier source.**
Toutes les fonctions dont la première instruction est associée à une ligne d'un fichier source donné.

**Classe.**
Les noms des symboles des fonctions sont généralement ordonnés hiérarchiquement dans des espaces de nommage, par exemple les espaces de nommage C++, ou les classes des langages orientés objet. Ainsi une classe peut contenir des fonctions d'une classe ou de classes embarquées.

**Partie d'un profilage.**
Quelques sections de temps d'une exécution de profilage, avec un identifiant de thread donné, un identifiant de processus, et la ligne de commande exécutée.

Comme on peut le voir depuis la liste, un ensemble d'entités de coût définit souvent une autre entité de coût. Ainsi il y a une hiérarchie d'imbrication des entités de coût qui semble évidente par rapport à la description faite au-dessus.

Tuples des positions :

— Appel d'une adresse d'instruction vers une fonction cible.

— Appel d'une ligne du source vers une fonction cible.

— Appel d'une fonction du source vers une fonction cible.

— Saut (in)conditionnel d'une source vers une instruction cible.

— Saut (in)conditionnel d'une source vers une ligne cible.

Les sauts entre les fonctions ne sont pas autorisés, car cela est absurde dans un arbre d'appels. Ainsi, les constructions telles la gestion des exceptions et les sauts longs en C doivent être traduits pour se mettre dans la pile d'appels comme demandé.

### 3.1.2 Types d'évènement

Des types d'évènements arbitraires peuvent être spécifiés dans les données de profilage en leur donnant un nom. Leur coût relié à une entité de coût est un entier sur 64 bits.

Les types d'évènement dont les coûts sont spécifiés dans le fichier de données de profilage sont appelés évènement réels. En plus, on peut spécifier des formules pour les types d'évènement calculés à partir d'évènement réels, que l'on appelle évènements hérités.

## 3.2 État de la vue

L'état de la vue de la fenêtre de KCachegrind inclut :

— le type d'évènement primaire et secondaire choisi pour l'affichage,

— le regroupement de fonction (utilisé dans la liste Profilage des fonctions et dans la coloration des entités),

— les parties du profilage dont les coûts doivent être inclus dans la vue,

— une entité active de coût (par exemple, une fonction sélectionnée de la barre latérale Profilage de la fonction),

— une entité de coût sélectionnée.

Cet état influence les vues.

Les vues sont toujours affichées pour une entité de coût, celle qui est active. Quand une vue donnée n'est pas appropriée pour une entité de coût, elle peut être désactivée (par exemple quand on sélectionne un objet ELF en double-cliquant dans la liste des groupes, l'annotation du code source pour un objet ELFne veut rien dire).

Par exemple, pour une fonction active, la liste des fonctions appelées montre toutes les fonctions appelées par la fonction active. On peut sélectionner chacune de ces fonctions sans la rendre active. Si le graphe d'appels est montré à côté, il va automatiquement sélectionner la même fonction.

## 3.3 Parties de l'interface graphique utilisateur

### 3.3.1 Barres latérales

Les barres latérales sont des fenêtres de côté qui peuvent être placées à chaque bordure de la fenêtre de KCachegrind. Elles contiennent toujours une liste d'entités de coût triées d'une manière quelconque.

— Le **profilage de la fonction**. Le profilage d'une fonction est une liste des fonctions avec les coûts inclusifs et exclusifs, le nombre d'appels, le nom et la position des fonctions.

— **Synthèse des parties**

— **Pile d'appel**

### 3.3.2 L'aire d'affichage

L'aire d'affichage, se situant généralement dans la partie droite de la fenêtre principale de KCachegrind, est constituée d'une (par défaut) ou de plusieurs vues d'onglets, rangées horizontalement ou verticalement. Chaque onglet contient plusieurs vues différentes pour une seule entité de coût à un instant donné. Le nom de cette entité est indiqué en haut de la vue d'onglets. S'il y a plusieurs vues d'onglets, seulement une est active. Le nom de l'entité dans la vue d'onglets active est affiché en gras et détermine l'entité de coût active de la fenêtre de KCachegrind.

### 3.3.3 Aires d'un d'onglet

Chaque vue d'onglet peut contenir jusqu'à quatre aires d'affichage, nommées Haut, Droite, Gauche, Bas. Chaque aire peut contenir plusieurs vues empilées. La vue visible d'une aire est sélectionnée par la barre d'onglets. Les barres d'onglets de l'aire en haut à droite sont en haut, les barres d'onglets de l'aire en bas à gauche sont en bas. Vous pouvez spécifier quel type de vue doit aller dans chaque aire en utilisant les menus contextuels des onglets.

### 3.3.4 Vue synchronisée par une entité sélectionnée dans une vue d'onglet.

En plus d'une entité active, chaque onglet a une entité sélectionnée. Comme la plupart des types de vues montre plusieurs entités avec celle qui est active centrée, vous pouvez changer l'élément sélectionné en naviguant dans une vue (en cliquant avec la souris ou en utilisant le clavier). Généralement, les éléments sélectionnés sont affichés en surbrillance. En changeant l'entité sélectionnée dans une des vues d'un onglet, toutes les autres vues mettent par conséquent la nouvelle entité sélectionnée en surbrillance.

### 3.3.5 Synchronisation entre les onglets

Si il y a plusieurs onglets, un changement de sélection dans un des onglets mène à un changement d'activation dans l'onglet suivant (à droite/en bas). Cette sorte de lien doit permettre, par exemple, de naviguer rapidement dans les graphes d'appels.

### 3.3.6 Dispositions

La disposition de toutes les vues d'onglets d'une fenêtre peut être enregistrée (**Vue** → **Disposition des vues**). Après avoir dupliqué la disposition courante (**Vue** → **Disposition des vues** → **Dupliquer** (**Ctrl++**)) et changé quelques tailles ou bougé une vue vers une autre aire

de la vue d'onglets, vous pouvez rapidement commuter entre la nouvelle disposition et l'ancienne par **Ctrl**+← et **Ctrl**+→. L'ensemble des dispositions sera enregistré entre les sessions de KCachegrind pour une même commande profilée. Vous pouvez faire que l'ensemble courant des dispositions soit celui par défaut pour les nouvelles sessions de KCachegrind, ou bien revenir à l'ensemble des dispositions par défaut.

## 3.4 Barres latérales

### 3.4.1 Profilage aplati

Le **profilage aplati** contient une liste de groupes et une liste de fonctions. La liste des groupes contient tous les groupes où le coût a été enregistré, en fonction du type de groupe choisi. La liste des groupes est cachée quand le regroupement est désactivé.

La liste des fonctions contient les fonctions d'un regroupement sélectionné (ou toutes les fonctions si le regroupement est désactivé), triées par colonne, par exemple les coûts propres ou inclusifs enregistrés dedans. Le nombre de fonctions affichées dans la liste est limité, mais configurable **Configuration** → **Configurer KCachegrind**..

### 3.4.2 Synthèse des parties

Dans une exécution de profilage, plusieurs fichiers de données de profilage peuvent être produits et être chargés ensemble dans KCachegrind. La barre latérale **Synthèse des parties**les montre, en les triant horizontalement par date de création, les tailles de rectangle étant proportionnelles au coût enregistré dans chaque partie. Vous pouvez sélectionner une ou plusieurs parties pour obliger les coûts affichés dans les autres vues de KCachegrind à s'appliquer uniquement sur ces parties.

Les parties sont encore divisées en un mode partitionnement et un mode partage des coûts inclusifs :

**Mode partitionnement**

Partitionnement : vous voyez le partitionnement dans des groupes pour une partie des données de profilage, en accord avec le type de groupe sélectionné. Par exemple, si les groupes objet ELFsont sélectionnés, vous verrez des rectangles colorés pour chaque objet ELF utilisé (bibliothèque partagée ou exécutable), qui auront une taille proportionnelle au coût enregistré dedans.

**Mode diagramme**

Un rectangle montrant le coût inclusif de la fonction active dans la partie est affiché. Celui-ci est partagé pour afficher les coûts inclusifs des fonctions appelées.

### 3.4.3 Pile d'appels

C'est une pile d'appels purement fictive, qui est la "plus probable". Elle est construite en mettant au début la fonction active courante, puis en ajoutant les fonctions appelantes/appelées avec les plus hauts coûts en haut et en bas.

Les colonnes **coût** et **appels** montrent le coût enregistré pour tous les appels de la fonction dans la ligne au-dessus.

## 3.5 Vues

### 3.5.1 Type d'évènement

La liste **Types** montre tous les types de coût disponibles, ceux correspondant, et le coût inclusif de la fonction active courante pour ce type d'évènement.

En choisissant un type d'évènement dans la liste, vous remplacez le type des coûts montré partout dans KCachegrind par celui sélectionné.

### 3.5.2 Listes des fonctions appelants

Ces listes montrent les appels et les fonctions appelées de la fonction active courante. **Toutes les fonctions appelantes** et **carte des fonctions appelées** désignent toutes les fonctions pouvant être accédées dans le sens des appelantes ou des appelées, même si d'autres fonctions se trouvent entre elles.
Les vues de liste des appels inclut :
— **Appelantes** directes
— **Appelées** directes
— **Toutes les fonctions appelantes**
— **Toutes les fonctions appelées**

### 3.5.3 Cartes

Une vue de la carte de l'arbre du type d'évènement primaire, en haut ou en bas de la hiérarchie d'appel. Chaque rectangle coloré représente une fonction, sa taille est approximativement proportionnelle au coût enregistré à l'intérieur pendant que la fonction active s'exécutait (cependant, il y a des contraintes de dessin).

Pour la **carte des fonctions appelantes**, le graphique montre la hiérarchie de toutes les fonctions appelant la fonction active courante ; pour la **carte des fonctions appelées**, il affiche ceci pour toutes les fonctions appelées.

Les options d'apparence sont disponibles dans le menu contextuel. Pour avoir des proportions exactes, choisissez **Cacher les bordures incorrectes**. Comme ce mode peut être très gourmand au niveau du temps, vous voudrez peut-être limiter avant le niveau maximum de dessin. **Meilleur** détermine la direction de partage pour les enfants à partir du ratio d'aspect de leur parent. **Toujours meilleur** décide de l'espace restant pour chaque enfant du même parent. **Ignorer les proportions** prend l'espace pour dessiner le nom de la fonction avant de dessiner les enfants. Notez que les proportions peuvent être fortement fausses.

La navigation par le clavier est disponible avec les touches gauche et droite pour parcourir les enfants du même parent, et haut et bas pour aller au niveau au-dessus et en dessous le plus proche. La touche **Entrée** active l'élément courant.

### 3.5.4 Graphe des appels

Cette vue montre le graphe d'appel autour de la fonction active. Le coût montré est seulement le coût enregistré pendant que la fonction active s'exécutait ; c'est-à-dire le coût montré pour `main()` - si elle est visible - doit être le même que le coût de la fonction active, comme c'est la partie du coût inclusif de `main()` enregistré pendant que la fonction active s'exécutait.

Pour les cycles, les flèches d'appels bleues indiquent que c'est un appel artificiel rajouté pour un affichage correct, même s'il ne s'est jamais produit.

Si le graphe est plus large que l'aire de représentation, un panneau d'aperçu est affiché dans un coin. Il y a des options de vues identiques à celles de la carte de l'arbre d'appels ; la fonction sélectionnée est mise en surbrillance.

### 3.5.5 Annotations

Les listes source/assembleur annoté montre les lignes du code source/les instructions désassemblées de la fonction active courante, ainsi que le coût (propre) enregistré lors de l'exécution du code de la ligne du source/l'instruction. S'il y a eu appel, les lignes avec les détails sur l'appel sont insérées dans le code source : le coût (inclusif) enregistré à l'intérieur de l'appel, le nombre d'appels effectués, et la destination de l'appel.

Sélectionnez une telle ligne d'information d'appel pour activer la destination de l'appel.

# Chapitre 4

# Les éléments de menus / barres d'outils

## 4.1 La fenêtre principale de KCachegrind

### 4.1.1 Le menu « Fichier »

**Fichier → Nouveau (Ctrl-N)**
Ouvre une fenêtre de haut niveau vide dans laquelle vous pouvez charger des données de profilage. Cette action n'est pas vraiment nécessaire, car **Fichier → Ouvrir** vous donnera une nouvelle fenêtre de haut niveau si la fenêtre courante affiche déjà des données.

**Fichier → Ouvrir (Ctrl-O)**
Affiche la boîte de dialogue de sélection de fichier de KDE afin que vous puissiez choisir le fichier de données de profilage à charger. S'il y a déjà des données affichées dans la fenêtre courante de haut niveau, ceci va ouvrir une nouvelle fenêtre. Si vous voulez ouvrir des données additionnelles de profilage dans la fenêtre courante, utilisez **Fichier → Ajouter**.

Le nom des fichiers de données de profilage se termine habituellement par `pid .partie-t hreadID`, où *`partie`* et *`threadID`* sont facultatifs. *`pid`* et *`partie`* sont utilisés pour les multiples fichiers de données de profilage recueillis lors de l'exécution d'une application. En chargeant un fichier se terminant seulement par`pid`, les fichiers de données éventuellement présents pour cette exécution, mais avec des terminaisons additionnelles, seront également chargés.

S'il existe des fichiers de données de profilage `cachegrind.out.123` et `cachegrind.out .123.1`, en chargeant le premier, le second sera chargé automatiquement.

**Fichier → Ajouter**
Ajoute un fichier de données de profilage dans la fenêtre courante. Vous pouvez ainsi forcer le chargement de multiples fichiers de données dans la même fenêtre de premier niveau, même s'ils ne sont pas de la même exécution, comme donné par la convention de nommage des fichiers de données de profilage. Par exemple, ceci peut être utilisé pour des comparaisons côte-à-côte.

**Fichier → Recharger (F5)**
Recharge les données de profilage. Ceci est utile après qu'un autre fichier de données de profilage ait été généré par l'exécution d'une application déjà chargée.

**Fichier → Quitter (Ctrl-Q)**
Quitte KCachegrind

# Chapitre 5

# Questions et Réponses

1. *À quoi sert KCachegrind ? Je n'en ai aucune idée.*
   KCachegrind est utile dans le stade final du développement d'un logiciel, appelé le profilage. Si vous ne développez pas d'applications, vous n'avez pas besoin de KCachegrind.
2. *Quelle est la différence entre* ***Incl.*** *et* ***Propre*** *?*
   Ce sont des attributs de coût pour les fonctions en considérant certains types d'évènements. Comme les fonctions peuvent s'appeler entre elles, il paraît logique de distinguer le coût de la fonction elle-même ("Coût propre") et le coût incluant toutes les fonctions appelées ("Coût inclusif"). " Propre " est aussi remplacé certaines fois par " Exclusif ".
   Ainsi, par exemple pour `main()`, vous aurez toujours un coût inclusif de presque 100 %, alors que le coût propre est négligeable, le travail réel s'effectuant dans une autre fonction.
3. *Si je double-clique sur une fonction en bas* ***Graphe des appels****, il affiche le même coût pour la fonction* `main()` *que pour la fonction sélectionnée. N'est-ce pas supposé rester constant à 100 % ?*
   Vous avez activé une fonction en dessous de `main()` avec un coût de tout évidence inférieur à celui de `main()` elle-même. Pour chaque fonction, on ne montre de la partie du coût total de la fonction, que celle enregistrée alors que la fonction *activée* s'exécutait. C'est-à-dire que le coût affiché pour toute fonction ne peut jamais être plus élevé que le coût de la fonction activée.

# Chapitre 6

# Glossaire

**Entité de coût**
: C'est un élément abstrait relié au code source, auquel on peut attribuer des compteurs d'évènements. Les dimensions des entités de coût sont la localisation dans le code (par exemple, ligne source, fonction), la localisation dans les données (par exemple, type de la donnée accédée, donnée), la localisation dans l'exécution (par exemple, thread, processus), et les couples ou les triplets des positions mentionnées au-dessus (par exemple, appels, accès à un objet à partir d'une instruction, donnée expulsée du cache).

**Coûts d'évènement**
: C'est la somme des évènements d'un type donné, se produisant pendant que l'exécution est reliée à une entité de coût donnée. Le coût est attribué à l'entité.

**Type d'évènement**
: Type d'évènement : c'est la sorte d'évènement dont les coûts peuvent être attribués à une entité de coût. Il existe des types d'évènements réels et des types d'évènements hérités.

**Types d'évènement hérités**
: C'est un type d'évènement virtuel, seulement visible dans une vue, et qui est défini par une formule utilisant des types d'évènements réels.

**Fichier de données de profilage**
: C'est un fichier contenant des données mesurées dans une expérience de profilage (ou une partie), ou produite par post-traitement d'une trace. Sa taille est généralement linéairement proportionnelle à la taille du code du programme.

**Données partielles de profilage**
: Données extraites d'un fichier de données de profilage.

**Expérience de profilage**
: C'est une exécution d'un programme, supervisée par un outil de profilage, qui peut produire plusieurs fichiers de données de profilage à partir de parties ou de threads de l'exécution.

**Projet de profilage**
: C'est une configuration pour les expériences de profilage utilisée pour un programme à profiler, peut-être dans plusieurs versions. Comparer des données de profilage n'a généralement de sens qu'entre données de profilage produites dans des expériences sur un seul projet de profilage.

**Profilage**
: C'est le processus de collecte d'informations statistiques sur les caractéristiques d'un programme qui s'exécute.

**Types d'évènement réel**
: C'est un type d'évènement qui peut être mesuré par un outil. Cela nécessite l'existence d'un capteur pour le type d'évènement donné.

**Trace**
: C'est une séquence d'évènements horodatés qui se produisent lors du traçage d'un programme qui s'exécute. Sa taille est généralement proportionnelle linéairement au temps d'exécution du programme.

**Trace partielle**
: Voir "Données partielles de profilage".

**Traçage**
: C'est le processus de supervision d'une exécution de programme, et de sauvegarde des évènements triés par date dans un fichier de sortie, la trace.

# Chapitre 7

# Remerciements et licence

Merci à Julian Seward pour son excellent outil Valgrind, et à Nicholas Nethercote pour le module externe Cachegrind. Sans ces programmes, KCachegrind n'aurait jamais existé. Ils sont par ailleurs à l'origine de beaucoup d'idées pour cette GUI.

Merci pour tous les rapports de bogues et les suggestions des différents utilisateurs.

Traduction française par Yann Verleyyann.verley@free.fr, Ludovic Grossard grossard@kde.org, Damien Raude-Morvan drazzib@drazzib.com et Joseph Richard jrchcell@gmail.com.