

# The KCachegrind Handbook

Josef Weidendorfer



# The KCachegrind Handbook

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Profiling . . . . .	1
1.2	Profiling Methods . . . . .	1
1.3	Profiling Tools . . . . .	2
1.4	Visualization . . . . .	3
<b>2</b>	<b>Using KCachegrind</b>	<b>5</b>
2.1	Generate Data to Visualize . . . . .	5
2.1.1	Callgrind . . . . .	5
2.1.2	OProfile . . . . .	6
2.2	User Interface Basics . . . . .	6
<b>3</b>	<b>Basic Concepts</b>	<b>8</b>
3.1	The Data Model for Profile Data . . . . .	8
3.1.1	Cost Entities . . . . .	8
3.1.2	Event Types . . . . .	9
3.2	Visualization State . . . . .	9
3.3	Parts of the GUI . . . . .	10
3.3.1	Sidedocks . . . . .	10
3.3.2	Visualization Area . . . . .	10
3.3.3	Areas of a Tab View . . . . .	11
3.3.4	Synchronized Visualization via Selected Entity in a Tab View . . . . .	11
3.3.5	Synchronization between Tab Views . . . . .	11
3.3.6	Layouts . . . . .	11
3.4	Sidedocks . . . . .	11
3.4.1	Flat Profile . . . . .	11

## The KCachegrind Handbook

3.4.2	Parts Overview . . . . .	12
3.4.3	Call Stack . . . . .	12
3.5	Visualizations . . . . .	12
3.5.1	Event Types . . . . .	12
3.5.2	Call Lists . . . . .	13
3.5.3	Maps . . . . .	13
3.5.4	Call Graph . . . . .	13
3.5.5	Annotations . . . . .	14
<b>4</b>	<b>Command Reference</b>	<b>15</b>
4.1	The main KCachegrind window . . . . .	15
4.1.1	The File Menu . . . . .	15
4.1.2	The View Menu . . . . .	16
<b>5</b>	<b>Questions and Answers</b>	<b>17</b>
<b>6</b>	<b>Glossary</b>	<b>18</b>
<b>7</b>	<b>Credits and License</b>	<b>20</b>
<b>A</b>	<b>Installation</b>	<b>21</b>
A.1	How to obtain KCachegrind . . . . .	21
A.2	Requirements . . . . .	21
A.3	Compilation and Installation . . . . .	21
A.4	Configuration . . . . .	21

### **Abstract**

KCachegrind is a profile data visualization tool, written using the KDE environment.

# Chapter 1

## Introduction

KCachegrind is a browser for data produced by profiling tools. This chapter explains what profiling is for, how it is done, and gives some examples of profiling tools available.

### 1.1 Profiling

When developing a program, one of the last steps often involves performance optimizations. As it makes no sense to optimize functions rarely used, because that would be a waste of time, one needs to know in which part of a program most of the time is spent.

For sequential code, collecting statistical data of the programs runtime characteristic like time numbers spent in functions and code lines usually is enough. This is called Profiling. The program is run under control of a profiling tool, which gives the summary of an execution run at the end. In contrast, for parallel code, performance problems typically are caused when one processor is waiting for data from another. As this waiting time usually cannot easily attributed, here it is better to generate timestamped event traces. KCachegrind cannot visualize this kind of data.

After analyzing the produced profile data, it should be easy to see the hot spots and bottlenecks of the code: for example, assumptions about call counts can be checked, and identified code regions can be optimized. Afterwards, the success of the optimization should be verified with another profile run.

### 1.2 Profiling Methods

To exactly measure the time passed or record the events happening during the execution of a code region (e.g. a function), additional measurement code

needs to be inserted before and after the given region. This code reads the time, or a global event count, and calculates differences. Thus, the original code has to be changed before execution. This is called instrumentation. Instrumentation can be done by the programmer itself, the compiler, or by the runtime system. As interesting regions usually are nested, the overhead of measurement always influences the measurement itself. Thus, instrumentation should be done selectively and results have to be interpreted with care. Of course, this makes performance analysis by exact measurement a very complex process.

Exact measurement is possible because of hardware counters (including counters incrementing on a time tick) provided in modern processors, which are incremented whenever an event is happening. As we want to attribute events to code regions, without the counters, we would have to handle every event by incrementing a counter for the current code region ourselves. Doing this in software is, of course, not possible; but, on the assumption that the event distribution over source code is similar when looking only at every  $n$ -th event instead of every event, a measurement method whose overhead is tunable has been developed: it is called Sampling. Time Based Sampling (TBS) uses a timer to regularly look at the program counter to create a histogram over the program code. Event Based Sampling (EBS) exploits the hardware counters of modern processors, and uses a mode where an interrupt handler is called on counter underflow to generate a histogram of the corresponding event distribution: in the handler, the event counter is always reinitialized to the ' $n$ ' of the sampling method. The advantage of sampling is that the code does not have to be changed, but it is still a compromise: the above assumption will be more correct if  $n$  is small, but the smaller the  $n$ , the higher the overhead of the interrupt handler.

Another measurement method is to simulate things happening in the computer system when executing a given code, i.e. execution driven simulation. The simulation is always derived from a more or less accurate machine model; however, with very detailed machine models, giving very close approximations to reality, the simulation time can be unacceptably high in practice. The advantage of simulation is that arbitrarily complex measurement/simulation code can be inserted in a given code without perturbing results. Doing this directly before execution (called runtime instrumentation), using the original binary, is very comfortable for the user: no re-compilation is necessary. Simulation becomes usable when simulating only parts of a machine with a simple model; another advantage is that the results produced by simple models are often easier to understand: often, the problem with real hardware is that results include overlapping effects from different parts of the machine.

### 1.3 Profiling Tools

Most known is the GCC profiling tool `gprof`: One needs to compile the program with option `-pg`; running the program generates a file `gmon.out`, which can be transformed into human-readable form with `gprof`. One disadvantage is the needed re-compilation step to prepare the executable, which has to be statically linked. The method used here is compiler-generated instrumentation -

which measures call arcs happening among functions and corresponding call counts - in conjunction with TBS - which gives a histogram of time distribution over the code. Using both pieces of information, it is possible to heuristically calculate inclusive time of functions, i.e. time spent in a function together with all functions called from it.

For exact measurement of events happening, libraries exist with functions able to read out hardware performance counters. Most known here is the PerfCtr patch for Linux®, and the architecture independent libraries PAPI and PCL. Still, exact measurement needs instrumentation of code, as stated above. Either one uses the libraries itself or uses automatic instrumentation systems like ADAPTOR (for FORTRAN source instrumentation) or DynaProf (code injection via DynInst).

OProfile is a system-wide profiling tool for Linux® using Sampling.

In many aspects, a comfortable way of Profiling is using Cachegrind or Callgrind, which are simulators using the runtime instrumentation framework Valgrind. Because there is no need to access hardware counters (often difficult with today's Linux® installations), and binaries to be profiled can be left unmodified, it is a good alternative to other profiling tools. The disadvantage of simulation - slowdown - can be reduced by doing the simulation on only the interesting program parts, and perhaps only on a few iterations of a loop. Without measurement/simulation instrumentation, Valgrind's usage only has a slowdown factor in the range of 3 to 5. Also, when only the call graph and call counts are of interest, the cache simulator can be switched off.

Cache simulation is the first step in approximating real times; as on modern systems, runtime is very sensitive to the exploitation of so called caches (small and fast buffers which accelerate repeated accesses to the same main memory cells.) Cachegrind does cache simulation by catching memory accesses. The data produced includes the number of instruction/data memory accesses and 1st/2nd level cache misses, and relates it to source lines and functions of the run program. By combining these miss counts, using miss latencies from typical processors, an estimation of spent time can be given.

Callgrind is an extension of Cachegrind that builds up the call graph of a program on-the-fly, i.e. how the functions call each other and how many events happen while running a function. Also, the profile data to be collected can be separated by threads and call chain contexts. It can provide profiling data on an instruction level to allow for annotation of disassembled code.

## 1.4 Visualization

Profiling tools typically produce a large amount of data. The wish to easily browse down and up the call graph, together with fast switching of the sorting mode of functions and display of different event types, motivates a GUI application to accomplish this task.

KCachegrind is a visualization for profile data fulfilling these wishes. Despite being programmed first with browsing the data from Cachegrind and Calltree in mind, there are converters available to be able to display profile data

## The KCachegrind Handbook

produced by other tools. In the appendix, a description of the Cachegrind/-Callgrind file format is given.

Besides a list of functions sorted according exclusive or inclusive cost metrics, and optionally grouped by source file, shared library or C++ class, KCachegrind features various visualization views for a selected function, namely

- a call-graph view, which shows a section of the call graph around the selected function,
- a tree-map view, which allows nested-call relations to be visualized, together with inclusive cost metric for fast visual detection of problematic functions,
- source code and disassembler annotation views, allowing to see details of cost related to source lines and assembler instructions.

## Chapter 2

# Using KCachegrind

### 2.1 Generate Data to Visualize

First, one wants to generate performance data by measuring aspects of the runtime characteristics of an application, using a profiling tool. KCachegrind itself does not include any profiling tool, but is good in being used together with Callgrind, and by using a converter, also can be used to visualize data produced with OProfile. Although the scope of this manual is not to document profiling with these tools, the next section provides short quickstart tutorials to get you started.

#### 2.1.1 Callgrind

Callgrind is a part of Valgrind, <http://valgrind.org>. Note that it previously was called Calltree, but that name was misleading.

Most common use is to prefix the command line to start your application with `valgrind --tool=callgrind`, like in

```
valgrind --tool=callgrind myprogram myargs
```

At program termination, a file `callgrind.out.pid` will be generated which can be loaded into KCachegrind.

More advanced use is to dump out profile data whenever a given function of your application is called. E.g. for **konqueror**, to see profile data only for rendering a web page, you could decide to dump the data whenever you select the menu item View/Reload. This corresponds to a call to `KonqMainWindow::slotReload`. Use

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

This will produce multiple profile data files with an additional sequential number at the end of the filename. A file without such a number at the end (only ending in the process PID) will also be produced; by loading this file into KCachegrind, all others are loaded too, and can be seen in the Parts Overview and Parts list.

### 2.1.2 OProfile

OProfile is available from <http://oprofile.sf.net>. Follow the installation instructions on the web site; but, before you do, check if your distribution does not already provide it as package (like SuSE®).

System-wide profiling is only permitted to the root user, as all actions on the system can be observed; therefore, the following has to be done as root. First, configure the profiling process, using the GUI **oprof\_start** or the command-line tool **opcontrol**. Standard configuration should be timer mode (TBS, see introduction). To start the measurement, run **opcontrol -s**. Then run the application you are interested in and, afterwards, do a **opcontrol -d**. This will write out the measurement results into files under folder `/var/lib/oprofile/samples/`. To be able to visualize the data in KCachegrind, do in an empty directory:

```
opreport -gdf | op2callgrind
```

This will produce a lot of files, one for every program which was running on the system. Each one can be loaded into KCachegrind on its own.

## 2.2 User Interface Basics

When starting KCachegrind with a profile data file as argument, or after loading one with File → Open..., you will see a navigation panel containing the function list at the left; and, on the right the main part, an area with visualizations for a selected function. This visualization area can be arbitrarily configured to show multiple visualizations at once.

At first start, this area will be divided into a top and a bottom part, each with different visualizations selectable by tabs. To move visualization views, use the context menu of the tabs, and adjust the splitters between visualizations. To quickly switch between different visualization layouts, use View/Layout/Duplicate, change the layout and switch between layouts with View/Layout/Next (or, even better, use the corresponding keyboard shortcuts).

The active event type is important for visualization: for Callgrind, this is, for example, Cache Misses or Cycle Estimation; for OProfile, this is "Timer" in the simplest case. You can change the event type via a combobox in the toolbar or in the Event Type view. A first overview of the runtime characteristics should be given when you select function main in the left list, and look at the call graph visualization; there, you see calls happening in your program. Note that

## The KCachegrind Handbook

the call graph view only shows functions with high event count. By double-clicking a function in the graph, it will change to show the called functions around the selected one.

To explore the GUI further, in addition to this manual, also have a look at the documentation section on the web site <http://kcachegrind.sf.net>. Also, every widget in KCachegrind has 'What's this' help.

## Chapter 3

# Basic Concepts

This chapter explains some concepts of the KCachegrind, and introduces terms used in the interface.

### 3.1 The Data Model for Profile Data

#### 3.1.1 Cost Entities

Cost counts of event types (like L2 Misses) are attributed to cost entities, which are items with relationship to source code or data structures of a given program. Cost entities not only can be simple code or data positions, but also position tuples. For example, a call has a source and a target, or a data address can have a data type and a code position where its allocation happened.

The cost entities known to KCachegrind are given in the following. Simple Positions:

- **Instruction.** An assembler instruction at a specified address.
- **Source Line of a Function.** All instructions that the compiler (via debug information) maps to a given source line specified by source file name and line number, and which are executed in the context of some function. The latter is needed because a source line inside of an inlined function can appear in the context of multiple functions. Instructions without any mapping to an actual source line are mapped to line number 0 in file "???".
- **Function.** All source lines of a given function make up the function itself. A function is specified by its name and its location in some binary object if available. The latter is needed because binary objects of a single program each can hold functions with the same name (these can be accessed e.g. with `dlopen/dlsym`; the runtime linker resolves functions in a given search order of binary objects used). If a profiling tool cannot detect the symbol name of a

function, e.g. because debug information is not available, either the address of the first executed instruction typically is used, or "???".

- Binary Object. All functions whose code is inside the range of a given binary object, either the main executable or a shared library.
- Source File. All functions whose first instruction is mapped to a line of the given source file.
- Class. Symbol names of functions typically are hierarchically ordered in name spaces, e.g. C++ namespaces, or classes of object oriented languages; thus, a class can hold functions of the class or embedded classes itself.
- Profile Part. Some time section of a profile run, with a given thread ID, process ID, and command line executed.

As can be seen from the list, a set of cost entities often defines another cost entity; thus, there is a inclusion hierarchy of cost entities which should be obvious from the description above.

Positions tuples:

- Call from instruction address to target function.
- Call from source line to target function.
- Call from source function to target function.
- (Un)conditional Jump from source to target instruction.
- (Un)conditional Jump from source to target line.

Jumps between functions are not allowed, as this makes no sense in a call graph; thus, constructs like exception handling and long jumps in C have to be translated to popping the call stack as needed.

### 3.1.2 Event Types

Arbitrary event types can be specified in the profile data by giving them a name. Their cost related to a cost entity is a 64-bit integer.

Event types whose costs are specified in a profile data file are called real events. Additionally, one can specify formulas for event types calculated from real events, which are called inherited events.

## 3.2 Visualization State

The Visualization state of a KCachegrind window includes:

- the primary and secondary event type chosen for display,

- the function grouping (used in the Function Profile list and entity coloring),
- the profile parts whose costs are to be included in visualization,
- an active cost entity (e.g. a function selected from the function profile dockable),
- a selected cost entity.

This state influences visualizations.

Visualizations always are shown for one, the active, cost entity. When a given visualization is not appropriate for a cost entity, it is disabled (e.g. when selecting an ELF object in the group list by double-clicking, source annotation for an ELF object make no sense).

For example, for an active function, the callee list shows all the functions called from the active one: one can select one of these functions without making it active; also, if the call-graph is shown nearside, it will automatically select the same function.

## 3.3 Parts of the GUI

### 3.3.1 Sidedocks

Sidedocks (Dockables) are side windows which can be placed at any border of a KCachegrind window. They always contain a list of cost entities sorted in some manner.

- Function Profile. The Function Profile is a list of functions showing inclusive and exclusive cost, call count, name and position of functions.
- Parts Overview
- Call Stack

### 3.3.2 Visualization Area

The visualization area, typically the right part of a KCachegrind main window, is made up of one (default) or more Tab Views, either lined up horizontally or vertically. Each tab view holds different visualization views of only one cost entity at a time. The name of this entity is given at the top of the tab view. If there are multiple tab views, only one is active. The entity name in the active tab view is shown in bold and determines the active cost entity of the KCachegrind window.

### 3.3.3 Areas of a Tab View

Each tab view can hold up to four view areas, namely Top, Right, Left, and Bottom. Each area can hold multiple stacked visualization views. The visible view of an area is selected by a tab bar. The tab bars of the top and right area are at the top; the tab bars of the left and bottom area are at the bottom. You can specify which kind of visualization should go into which area by using the context menus of the tabs.

### 3.3.4 Synchronized Visualization via Selected Entity in a Tab View

Besides an active entity, each tab view has a selected entity. As most visualization types show multiple entities with the active one somehow centered, you can change the selected item by navigating inside a visualization (by clicking with the mouse or using the keyboard). Typically, selected items are shown in a highlighted state. By changing the selected entity in one of the visualizations of a tab view, all other visualizations in the tab view accordingly highlight the new selected entity.

### 3.3.5 Synchronization between Tab Views

If there are multiple tab views, a selection change in one tab view leads to an activation change in the next (to right/to bottom) tab view. This kind of linkage should, for example, allow for fast browsing in call graphs.

### 3.3.6 Layouts

The layout of all the tab views of a window can be saved (see menu item View/Layout). After duplicating the current layout (Ctrl+Plus or menu) and changing some sizes or moving a visualization view to another area of a tab view, you can quickly switch between the old and the new layout via Ctrl+Left/Right. The set of layouts will be stored between KCachegrind sessions of the same profiled command. You can make the current set of layouts as the default one for new KCachegrind sessions, or restore to the default layout set.

## 3.4 Sidedocks

### 3.4.1 Flat Profile

The flat profile contains a group list and a function list. The group list contains all groups where cost is spent in, depending on the chosen group type. The group list is hidden when grouping is switched off.

The function list contains the functions of the selected group (or all functions if grouping is switched off), ordered by some column, e.g. inclusive or self costs spent therein. There is a maximal number of functions shown in the list, which is configurable in Settings/Configure KCachegrind.

### 3.4.2 Parts Overview

In a profile run, multiple profile data files can be produced, which can be loaded together into KCachegrind. The Parts Overview dockable shows these, horizontally ordered according creation time; the rectangle sizes are proportional to the cost spent in the parts. You can select one or several parts to constrain the costs shown in the other KCachegrind views to these parts only.

The parts are further subdivided: there is a partitioning and an inclusive cost split mode:

- Partitioning: You see the partitioning into groups for a profile data part, according to the group type selected. For example, if ELF object groups are selected, you see colored rectangles for each used ELF object (shared library or executable), sized according to the cost spent therein.
- Inclusive Cost Split: A rectangle showing the inclusive cost of the current active function in the part is shown. This, again, is split up to show inclusive costs of its callees.

### 3.4.3 Call Stack

This is a purely fictional 'most probable' call stack. It is built up by starting with the current active function and adds the callers/callees with highest cost at the top and to bottom.

The 'Cost' and 'Calls' columns show the cost used for all calls from the function in the line above.

## 3.5 Visualizations

### 3.5.1 Event Types

This list shows all cost types available and the corresponding self and inclusive cost of the current active function for that event type.

By choosing an event type from the list, you change the type of costs shown all over KCachegrind to be the selected one.

### 3.5.2 Call Lists

These lists show calls to/from the current active function. With 'all' callers/callees functions are meant which can be reached in caller/callee direction, even when other functions are in between.

Call list views include:

- Direct Callers
- Direct Callees
- All Callers
- All Callees

### 3.5.3 Maps

A treemap visualization of the primary event type, up or down the call hierarchy. Each colored rectangle represents a function; its size tries to be proportional to the cost spent therein while the active function is running (however, there are drawing constraints).

For the Caller Map, the graph shows the nested hierarchy of all callers of the current activated function; for the Callee Map, it shows the nested hierarchy of all callees of the current activated function.

Appearance options can be found in the context menu. To get exact size proportions, choose 'Hide incorrect borders'. As this mode can be very time consuming, you may want to limit the maximum drawn nesting level before. 'Best' determinates the split direction for children from the aspect ratio of the parent. 'Always Best' decides on remaining space for each sibling. 'Ignore Proportions' takes space for function name drawing before drawing children. Note that size proportions can get heavily wrong.

Keyboard navigation is available with the left/right arrow keys for traversing siblings, and up/down arrow keys to go a nesting level up/down. 'Return' activates the current item.

### 3.5.4 Call Graph

This view shows the call graph around the active function. The shown cost is only the cost which is spent while the active function was actually running; i.e. the cost shown for `main()` - if it's visible - should be the same as the cost of the active function, as that is the part of inclusive cost of `main()` spent while the active function was running.

For cycles, blue call arrows indicate that this is an artificial call added for correct drawing which actually never happened.

If the graph is larger than the widget area, a bird's eye view is shown in one edge. There are similar visualization options as for the Call Treemap; the selected function is highlighted.

### 3.5.5 Annotations

The annotated source/assembler lists show the source lines/disassembled instructions of the current active function together with (self) cost spent while executing the code of a source line/instruction. If there was a call, lines with details on the call are inserted into the source: the (inclusive) cost spent inside of the call, the number of calls happening, and the call destination.

Select such a call information line to activate the call destination.

## Chapter 4

# Command Reference

### 4.1 The main KCachegrind window

#### 4.1.1 The File Menu

**File** → **New (Ctrl-N)** Opens an empty toplevel window into which you can load profile data. This action is not really needed, as **File** → **Open...** will give you a new toplevel window when the current one shows already some data.

**File** → **Open (Ctrl-O)** Pops up the File Open Dialog to choose a profile data file to be loaded. If there is some data already shown in the current toplevel window, this will open a new window; if you want to open additional profile data in the current window, use **File** → **Add...**

The name of profile data files usually ends in `'pid.part-threadID'`, where `'part'` and `'threadID'` are optional `'pid'` and `'part'` are used for multiple profile data files belonging to one application run. By loading a file ending only in `'pid'`, eventually existing data files for this run, but with additional endings, are loaded too.

Example: If there exist profile data files `cachegrind.out.123` and `cachegrind.out.123.1`, by loading the first, the second will be automatically loaded too.

**File** → **Add...** Adds a profile data file to the current window. Using this, you can force multiple data files to be loaded into the same toplevel window even if they are not from the same run as given by the profile data file naming convention. This can, for example, be used for nearside comparison.

**File** → **Reload** Reload the profile data. This is most interesting after another profile data file was generated for an already loaded application run.

**File** → **Quit (Ctrl-Q)** Quits KCachegrind

### 4.1.2 The View Menu

**View** → **Primary Event Type** (To-do)

**View** → **Secondary Event Type** (To-do)

**View** → **Grouping** (To-do)

**View** → **Layout** (To-do)

**View** → **Split** (To-do)

## Chapter 5

# Questions and Answers

This document may have been updated since your installation. You can find the latest version at <http://docs.kde.org/>.

1. *What is KCachegrind for? I have no idea.*

KCachegrind is helpful at a later stage in software development, called Profiling. If you don't develop applications, you don't need KCachegrind.

2. *What is the difference between 'Incl.' and 'Self' ?*

These are cost attributes for functions regarding some event type. As functions can call each other, it makes sense to distinguish the cost of the function itself ('Self Cost') and the cost including all called functions ('Inclusive Cost'). 'Self' is sometimes also referred to as 'Exclusive' costs. So, for example, for `main()`, you will always have an inclusive cost of almost 100%, whereas the self cost is negligible when the real work is done in another function.

3. *The toolbar/menubar of my KCachegrind looks so spartan. Is this normal?*

Obviously KCachegrind is wrongly installed on your system. It is recommended to compile it with the installation prefix to be your system wide KDE base folder like **`configure --prefix=/opt/kde4; make install`**. If you choose another folder like `$HOME/kde`, you should set the environment variable `KDEDIR` to this folder before running KCachegrind.

4. *If I double-click on a function down in the Call Graph View, it shows for the function main the same cost as the selected function. Isn't this supposed to be constant 100% ?*

You have activated a function below `main()` with a cost less than `main()`. For any function, only that part of the full cost of the function is shown, that is spent while the activated function is running; that is, the cost shown for any function can never be higher than the cost of the activated function.

## Chapter 6

# Glossary

The following is a mixed list of terms.

- **Profiling:** The process of collecting statistical information about runtime characteristics of program runs.
- **Tracing:** The process of supervising a program run and storing events happening sorted by a timestamp into an output file, the Trace.
- **Trace:** A sequence of timestamped events that occurred while tracing a program run. Its size is typically linear to the execution time of the program run.
- **Profile Data File:** A file containing data measured in a profile experiment (or part of) or produced by postprocessing a trace. Its size is typically linear to the code size of the program.
- **Profile Data Part (incorrectly used also: Trace Part):** Data from a profile data file.
- **Profile Experiment:** A program run supervised by a profiling tool, producing possibly multiple profile data files from parts and/or threads of the run.
- **Profile Project:** A configuration for profile experiments used for one program which has to be profiled, perhaps in multiple versions. Comparisons of profile data typically only makes sense between profile data produced in experiments of one profile project.
- **Cost Entity:** An abstract item related to source code to which event counts can be attributed. Dimensions for cost entities are code location (e.g. source line, function), data location (e.g. accessed data type, data object), execution location (e.g. thread, process), and tuples or triples of the aforementioned positions (e.g. calls, object access from statement, evicted data from cache).
- **Event Type:** The kind of event of which costs can be attributed to a cost entity. There are real event types and inherited event types.

## The KCachegrind Handbook

- Real Event Type: An event type that can be measured by a tool. This needs the existence of a sensor for the given event type.
- Inherited Event Type: A virtual event type only visible in the visualization which is defined by a formula to be calculated from real event types.
- Event Costs: Sum of events of some event type occurring while the execution is related to some cost entity. The cost is attributed to the entity.

## Chapter 7

# Credits and License

KCachegrind

Thanks to Julian Seward for his excellent Valgrind, and Nicholas Nethercote for the Cachegrind addition. Without these programs, KCachegrind would not exist. Some ideas for this GUI were from them, too.

And thanks for all the bug reports/suggestions from different users.

This documentation is licensed under the terms of the [GNU Free Documentation License](#).

## Appendix A

# Installation

### A.1 How to obtain KCachegrind

KCachegrind is part of the kdesdk package of KDE. For less supported interim releases, Callgrind and further documentation, see the homepage at <http://kcachegrind.sf.net>. Look there for further installation and compile instructions.

### A.2 Requirements

In order to successfully use KCachegrind, you need KDE 4.x. For generating profile data, Cachegrind or Calltree/Callgrind is recommend.

### A.3 Compilation and Installation

For detailed information on how to compile and install KDE applications see [Building KDE4 From Source](#)

Since KDE uses **cmake** you should have no trouble compiling it. Should you run into problems please report them to the KDE mailing lists.

### A.4 Configuration

All configuration options are either in the configuration dialog or in the context popup menus of the visualizations.