

C++ development tools for Linux

7. Juni 2004

Note légale

Dieser Beitrag ist lizenziert unter der GNU Free Documentation License.

Note légale

Dieser Beitrag ist lizenziert unter der GNU General Public License.

Zusammenfassung

If you are a hobby carpenter, then you might very well get by with a screwdriver only. A screwdriver can be used to drive screws into a wall, you can use it to put nails in the wall, and even to get them out again if you are just a bit of a handy man. If you are a professional carpenter, then you do indeed need other and more specific tools than just a screwdriver.

Likewise a programmer doing assignments in toy programs might get by with something as simple as `printf` statements, while the more serious programmer needs better tools.

In this presentation, Jesper Pedersen will display his toolbox for professional software development. The toolbox include tools like: - `valgrind` - a tool for debugging memory errors - `cachegrind` - a tool for profiling applications. - `icecream` - a tool for parallel compilation. - `gdb` - The debugger - `xemacs` - an editor highly optimized for software development, including packages like `power-macros`, `wide-edit` and `klaralv.el`. - version control systems like `CVS`.

This presentation will mainly focus on the debugging tools `valgrind`, `cachegrind`, and `gdb`, while it will introduce the listener to the other applications.

1 Tools in general

1.1 C++ development tools for Linux

If you are a hobby carpenter, then you might very well get by with a screwdriver only. A screwdriver can be used to drive screws into a wall, you can use it to put nails in the wall, and even to get them out again if you are just a bit of a handy man. If you are a professional carpenter, then you do indeed need other and more specific tools than just a screwdriver.

Likewise a programmer doing assignments in toy programs might get by with something as simple as `printf` statements, while the more serious programmer needs better tools.

This article is split in two parts. In first part I will introduce you to some of the common development tools which you may want to use for software development, these include development environment, editors, debuggers, version control systems, and systems for speeding up compilations. In the second part I will focus on some pretty advanced tools for finding memory errors, tracing memory usage, plus finding where time is wasted in your application. All these tools are based on an open source tool called `valgrind`

1.2 Choosing the development environment

As part of my daytime job I'm giving courses in Qt programming <<http://www.klaralvdalens-datakonsult.se/?page=courses>>, and have consequently seen a lot of people developing software, the environment in which they do so, etc.

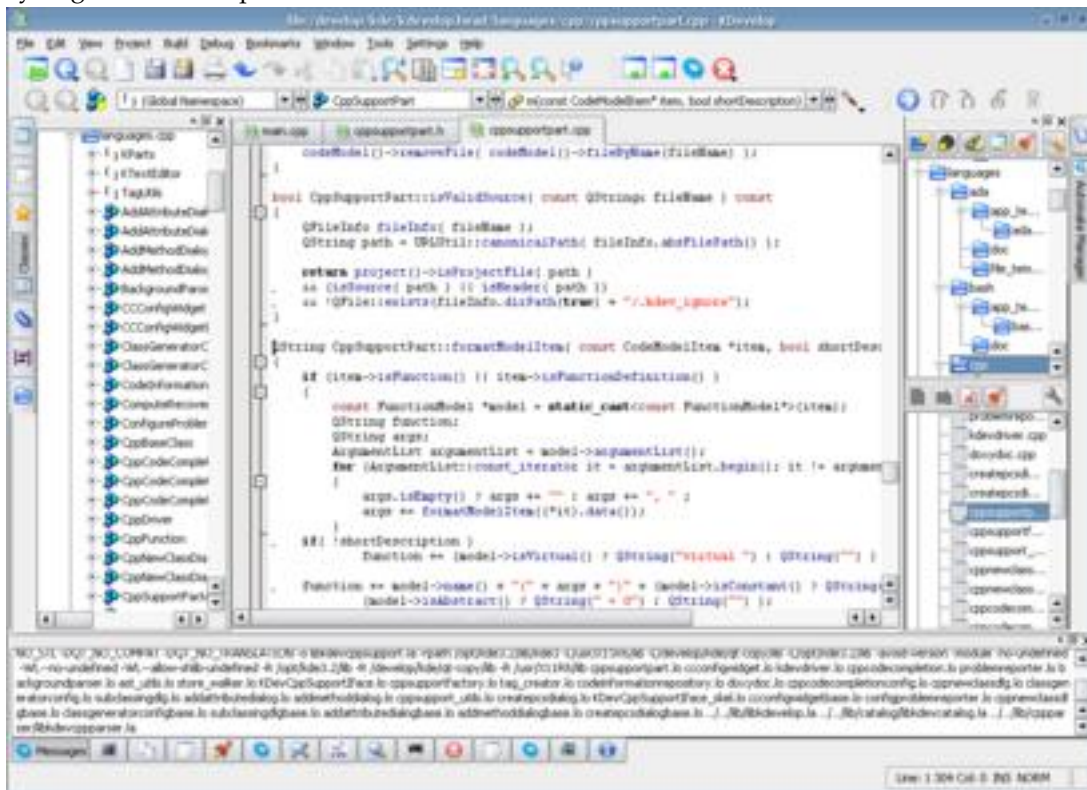
Sometimes, I've seen people who would start `vi`, make some changes, quit it again, and compile in the shell, sees a syntax error reported from `gcc`, start `vi` again, and makes the changes, and so on. For each error

they would spend 10 seconds to find the right file, find the right line and try and remember what the error was. To say the least this is not very efficient at all.

Thus when developing software, you have to make sure that your development environment is powerful enough to support your development cycle, both short term (as in compile cycle), and long term (as in merging conflicts resulting from several developers working on the same files).

People often ask me then, which editor they should choose, and even though my favorite editor is by far XEmacs, I don't want to get into any religious wars here. Choose the editor which you know, and become an expert with it - of course if your favorite editor is a simple and limited one, then you may consider changing.

Here are a few suggestions though. If you do not have a favorite editor already, then do consider KDevelop (see screen shot below), it is very powerful, very intuitive, and is being actively developed. If your favorite editor is Vim or one of the Emacs's, just fine, they are powerful enough for software development, just ensure that you get to be an expert with them



Remember the old story about two men cutting wood. One worked hard all day, while the other from time to time took a break. At the end of the day the person taking breaks would have - to the surprise of the other - cut more wood. When asked how that could be, he answered: "Every time I took a break, I sharpened my saw"

I urge you to think of your development environment as your *saw*, and from time to time spent some time learning about your development environment.

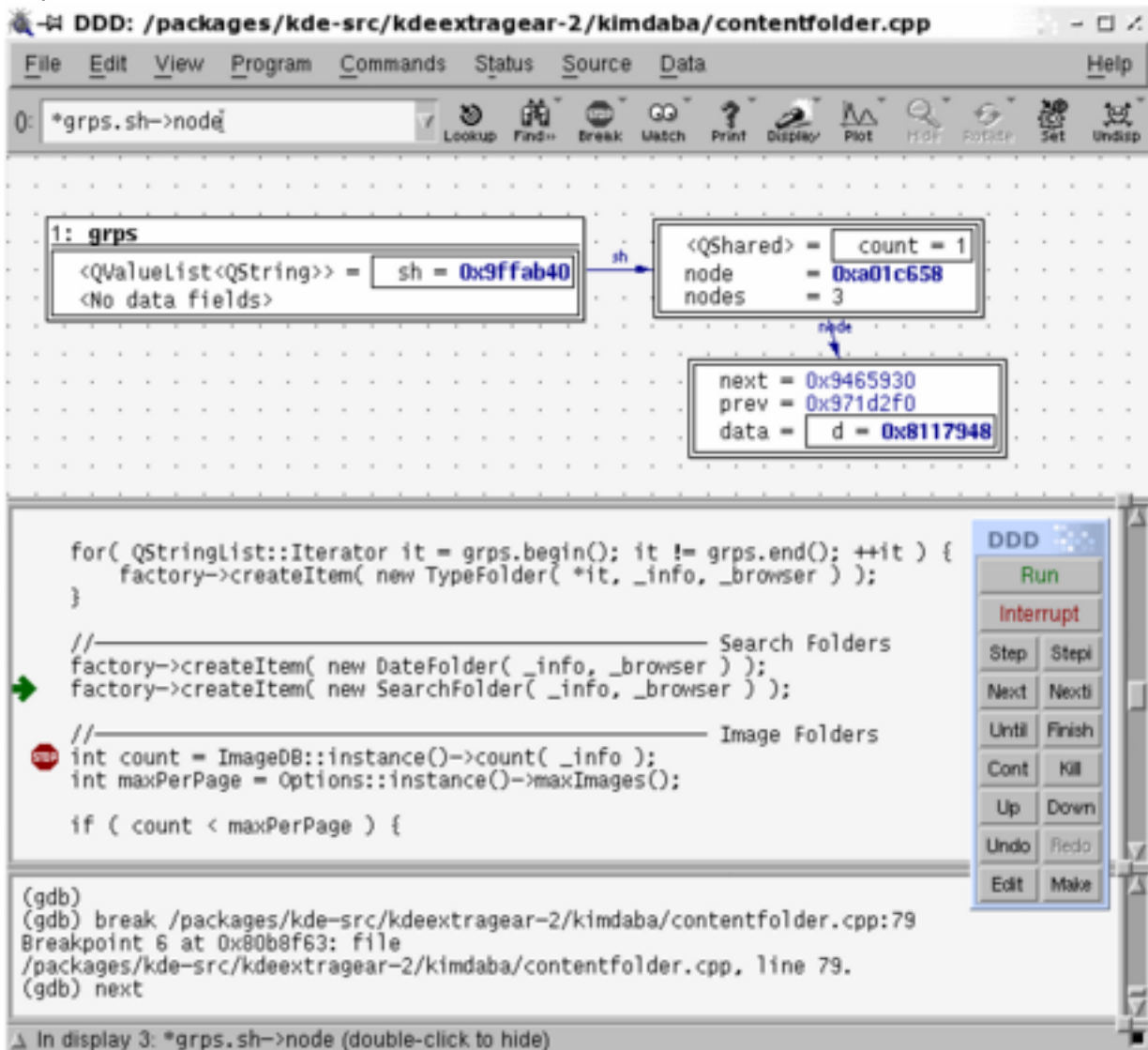
1.3 The Debugger

Unfortunately there are not much competitions for debuggers under Linux, which might also be the reason for the one and only existing debugger being only semi-good. For me personally it crashes up to tens times a day on busy debugging days. There are commercial debugger available for Linux, but as they are pretty expensive they are indeed only for professionals.

The debugger under Linux is called GDB, if you prefer you can run it directly from the command line. KDevelop has builtin support for GDB, here you simply click on a line to set a break point, and when execution stops at a break point, the editor will display that line for you.

Similar functionality exists for Emacs, with a bit more rough edge of course (setting breakpoints does not happen by clicking on the line, but rather by pressing C-x space). Vim has a patch for this (called VimGDB), but otherwise you might choose to use another front end for GDB.

There does exist a few graphical front ends to GDB, the most prominent one being DDD. This might be a good candidate for debugging if you do not use KDevelop (which has similar functionality to DDD built in). Below you can see a screen shot of DDD in action.



Talking about running gdb from within Emacs or VIM, both also has support to run the compiler from within the editor. Under Emacs simply press `M-x compile`, while under VIM press `:make`. Under Emacs you jump to the next error by pressing `C-x `` (i.e. the back ping), while under VIM you press `:cnext`.

1.4 Version Control Systems

When many people (that is more than one person) works together on a set of source code, it is of vital importance that the source files are managed by a revision system, which will handle the situation that two persons edits the same file.

Of course, an other important role for a version control system is the capabilities to figure out who changed what and when; being able to tag a number of files for release so you later can get back to that exact version, etc.

Several version control systems exists under Linux, the most widespread one is called CVS. It has some annoyances, but is stable like rocks (To my knowledge we never had much trouble with it in KDE).

Several free version control systems are actively being developed to replace CVS as the number one under Linux, some of these are `subversion` and `arch`. A number of commercial system also exists, among these are `bitkeeper`, `perforce`, `SourceSafe`, and `ClearCase`.

1.5 Speeding up compilations

The best way to get your applications compiled fast is to buy an ultra fast multi processor machine with hundreds of processors.

Not an options? Well then the next best and for most mortals most feasible solution is to have a bunch of computers work together on compiling your application. Three tools exists for this:

- `teambuilder` which is a commercial tool from Trolltech (the authors of Qt). See <http://www.trolltech.com/> <<http://www.trolltech.com/>> .
- `distcc` - an open source project. See <http://distcc.samba.org/> <<http://distcc.samba.org/>> .
- `icecream` - an open source project similar to `teambuilder` but building on `distcc`. (Not yet released, but can be downloaded from kde-nonbeta CVS. See <http://developer.kde.org/source/index.html> <<http://developer.kde.org/source/index.html>>)

`Teambuilder` and `icecream` are similar in functionality, with the only difference that `teambuilder` is more stable and commercial, while `icecream` is open source software. `Distcc` is a bit more stable than `icecream`, but its scheduling algorithm is round robin, where `icecream` and `teambuilder` tries to allocate the jobs to the best machine.

Having mentioned that `icecream` is not as stable as the others, its time to point out that we are not talking mission critical stuff here, so the scheduler crashing a few times a day is indeed not a disaster, and by the time you read this, the bugs are likely gone already, and its stable like rocks.

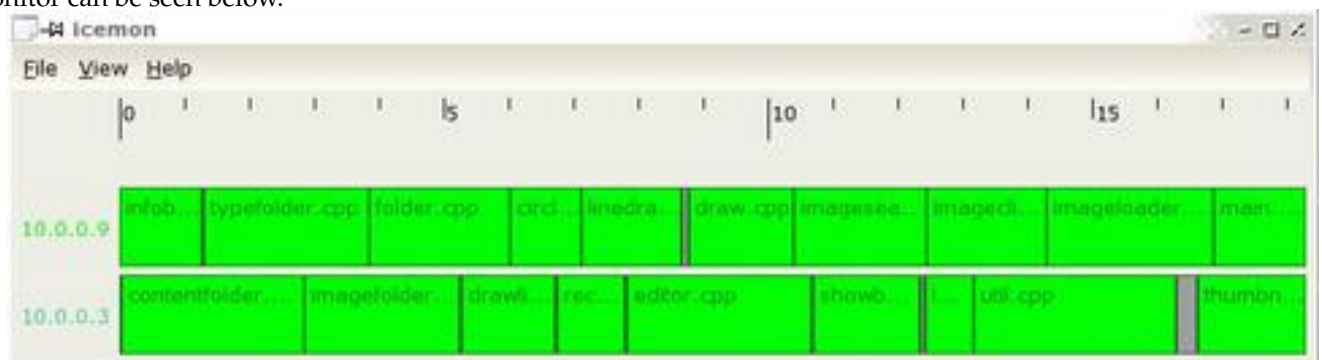
I'll therefore tell you a bit about `icecream`, as for most purposes this seems to be to be the best tool.

`Icecream` is very easy to use, here is how. On one machine you start the scheduler, and on all machines which should participate, you prepend the `icecream` directory to your path, which will ensure that the special versions of `gcc`, `cc` etc are in front of the normal ones. Next you must start the `icecream` daemon called `iceccd` . The compiler must be the same on all participating machines, but there are no requirements about having the same set of header files or libraries.

New machines can be put into the network without having to restart anything, and at any point any machine (except the scheduler) can be removed.

When compiling using `make`, you must add the compile flag `-j` to ensure that `make` issues more than one `gcc` compilation at a time. `-j` takes as an argument a maximum of simultaneous compilations, so if you have 5 machines participating, it might be an idea to specify `-j 5` , to avoid that `make` prepares say 200 files to compile, just to realize that there was an error in the first one.

`Icecream` comes with a monitor, which can show you which machines is doing what. A screen dump of the monitor can be seen below.



2 Advanced Development Tools

Juli 30th 2002 a tool called `valgrind` was released in version 1.0.0. If you should build a top 5 of date that changed history for software development under Linux, this would indeed include this date!

Until `valgrind` came into existence, open source software developers could just dream of having licenses to commercial tools like `purify` , `Insure++` , and `BoundsChecker` . At that time our best bet would probably be tools like `electric fence` which indeed was much harder to work with plus less powerful.

Valgrind is a modular toolkit from which two major tools has been developed plus a number of minor tools. In this part we will have a look at memcheck which is used to find memory errors, calltree which is used to profile applications, and massif which can be used to figure out which part of your application uses how much memory through the life time of the application.

Valgrind is an open source software which indeed is not a toy. It has been executed on any major C and C++ application you can think of in the Linux world, including KDE, Gnome, OpenOffice etc. The only catch there is to valgrind is that when you run an application under the control of valgrind then the application will execute in the order of magnitude 5-100 times slower.

2.1 The mem-check tool

As mentioned above, valgrind is a modular toolkit from which a number of tools are build. The fundamentals of valgrind is an X86 simulator, which simulates a real processor, and meanwhile measures certain things on your program.

The first tool we will look at is mem-check, a tool for tracing memory errors. Lets start with a small example from a very simple C program:

```
1
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main( int /*argc*/, char** /*argv*/ ) {
8     char* str = "Hello World\n";
9     char* copy = (char*) malloc( sizeof(char) * strlen( str ) );
10
11     strcpy( copy, str );
12 }
13
```

Take some time to look at the application above, if you can't spot the error within a millisecond, and if you think you could ever have made that error (and you could), then mem-check is indeed a tool for you.

The problem with the code above is that we allocate one byte too little for the copy of the string, namely the byte for the null terminating character. Nevertheless the strcpy function will copy over the null terminating character, and the result is that it writes beyond the boundary of the array allocated.

It is exactly this kind of problem that the mem-check tool for valgrind is created to detect (among a number of others of course). So lets try and see what valgrind says when invoked on the program above. To start valgrind with the mem-check tool, write a command similar to: `valgrind --tool=memcheck < application-name >` Below you can see the output from valgrind (Notice versions of valgrind older than 2.1.0 uses `--skin` rather than `--tool`)

```
1
2
3 ==19989== Memcheck, a memory error detector for x86-linux.
4 ==19989== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward.
5 ==19989== Using valgrind-2.1.1, a program supervision framework for x86-linux.
6 ==19989== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward.
7 ==19989== For more details, rerun with: -v
8 ==19989==
9 ==19989== Invalid write of size 1
10 ==19989==    at 0x3C020CAA: strcpy (mac_replace_strmem.c:199)
11 ==19989==    by 0x804855A: main (main.cpp:9)
12 ==19989== Address 0x3CE29060 is 0 bytes after a block of size 12 alloc'd
13 ==19989==    at 0x3C021350: malloc (vg_replace_malloc.c:105)
14 ==19989==    by 0x8048546: main (main.cpp:7)
```

```

15 ==19989==
16 ==19989== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 49 from 1)
17 ==19989== malloc/free: in use at exit: 360 bytes in 11 blocks.
18 ==19989== malloc/free: 404 allocs, 393 frees, 9422 bytes allocated.
19 ==19989== For a detailed leak analysis, rerun with: --leak-check=yes
20 ==19989== For counts of detected errors, rerun with: -v
21

```

Each line is prefixed with the process id of the processes printing out messages - this is useful as valgrind is capable of tracing subprocesses too (using the `--trace-children=yes` option).

Line 7 is where all the fun starts. Valgrind tells us that there has been an invalid write of one byte. The following lines are a back trace which shows up where this error occurred. But that's not all, on line 10 it tells us that the invalid write is zero bytes after another block, and then follows the backtrace telling us where that other block was allocated. Now it is a piece of cake finding and solving the problem.

2.1.1 What can mem-check check?

The following list is from the documentation of mem-check telling what it is capable of checking: (for details see the valgrind documentation.)

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – where pointers to malloc'd blocks are lost forever
- Passing of uninitialised and/or unaddressible memory to system calls
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

2.1.2 Suppressing errors

Valgrind does not only run your part of the application in its virtual machine, it runs the complete application, which for a KDE application for example includes KDE libraries, the Qt library, X11 libraries, the C library etc. Even though you may feel that your whole Linux box is pretty stable, a number of more or less harmful errors exists in the aforementioned libraries, which likely is out of your control to fix anyway.

Valgrind therefore has the capability of ignoring certain errors using ignore filters. By default a number of ignore filters are already in action when you start the mem-check tool, but you may also create your own filters, in the not-so-unlikely event that you are using a third party library which contains errors valgrind doesn't know about, or even in the situation that the project on which you work are so large that you can not make every single part of it valgrind clean.

Suppression rules are not to be written by mere mortals, so fortunately valgrind is capable of creating them for you, simply start valgrind using the flag `--gen-suppressions=yes`. It will then ask you if it should build a rule for you each time it sees an error. Cut'n'paste the rules you want into a file, and next time you start valgrind use `--suppressions= <filename >`

2.1.3 Starting GDB from valgrind

When mem-check encounters an error it is capable of starting gdb for you so you can introspect your application in the exact situation where the error is met. To make that possible simply start valgrind using the flag `-gdb-attach=yes`

Once mem-check encounters an error it will ask you if it should start gdb. Once you are done looking at the error in gdb, you must quit gdb with the following command: `detach` followed by `quit`. This will ensure that execution continues in the virtual processor. In other words do not invoke `continue` from within gdb.

2.1.4 Using mem-check for detecting memory leaks

The final feature of mem-check I'd like to highlight here is its capability to detect memory leaks in your application. Simply start valgrind using the option `--leak-check=yes`, and it will print a report on exit of application.

If you want to get your application 100% leak-free, a good idea is to ensure that all memory allocated is deleted, including that allocated in `main()`. Doing so, you may add the flag `--show-reachable=yes`, which will make mem-check report on memory to which there still is a valid pointer upon program termination.

2.1.5 addrcheck

While developing valgrind, the author set out to give valgrind a good exercise, he would run all of his KDE environment under valgrind. After having waited for a long time, he realized that valgrind was simply too slow for running a complete desktop environment for several days. What he did was therefore to cut some features of from valgrind to make it faster. The result is the tool `addrcheck` which is about twice as fast as valgrind, but does not check for use of uninitialized variables.

2.2 The calltree tool

Above we discussed the valgrind tool called `memcheck` and `addrcheck`, in the following we will look at another tool called `calltree`. `Calltree` was initially developed as a tool to analyze cache misses (cache as the fast memory computers have). It was, however, soon discovered that the data that `calltree` collected was also useful for a much broader audience. `calltree` output is namely useful for analysis of where time is spent in an application - which is also known as profiling.

`calltree`'s output is viewed using an application called `KCacheGrind`. The procedure for profiling is to start the valgrind using a command similar to `valgrind --tool=calltree <application>` This will generate one or more files named `cachegrind.out.<process-id>`. Next you will start `KCacheGrind` giving it one of these file as argument.

`calltree` and `kcachegrind` are not shipped with valgrind, but must instead be downloaded and installed separately. You download `calltree` and `KCacheGrind` from <http://kcachegrind.sourceforge.net/> <<http://kcachegrind.sourceforge.net/>>.

2.2.1 Controlling when dumps are made

There are two golden rules when it comes to profiling: First, a human has very seldom a clue at all where in the code time are spent (That's why you want to use tools like `calltree`). Second you should never optimize code unless the application feels slow. Thus when you want to profile your application, you should first figure out where it feels slow, and spent your energy on optimizing that part. Optimizations always leads to code that is harder to read and maintain so over-optimizing is not a good idea.

The application `ct_control` is used to control a running `calltree` tool. It basically has two options `-z` and `-d`. `-z` tells `calltree` to zero all its collections while `-d` tells it to dump its current information to file.

So in the situation where you are profiling a graphical user interface, and know that when you select, say a certain menu item from the menu bar, then an action is performed, which takes to long, here are the steps to take: Start your application under `calltree`'s control, do whatever you need to do to get to the situation where something is to slow. In a shell execute `ct_control -z`, this will make `calltree` behave as if it was just started at this point. Now select the action which feels too slow, and when it is done run `ct_control -d`, this will

make calltree dump whatever information it has collected since the start of the action. Now you may finish the program or even kill it if you want.

In the situation where you by other means have reached the conclusion that a given function is too slow, and want to optimize that, you may control when calltree zeros and dumps information using command line options.

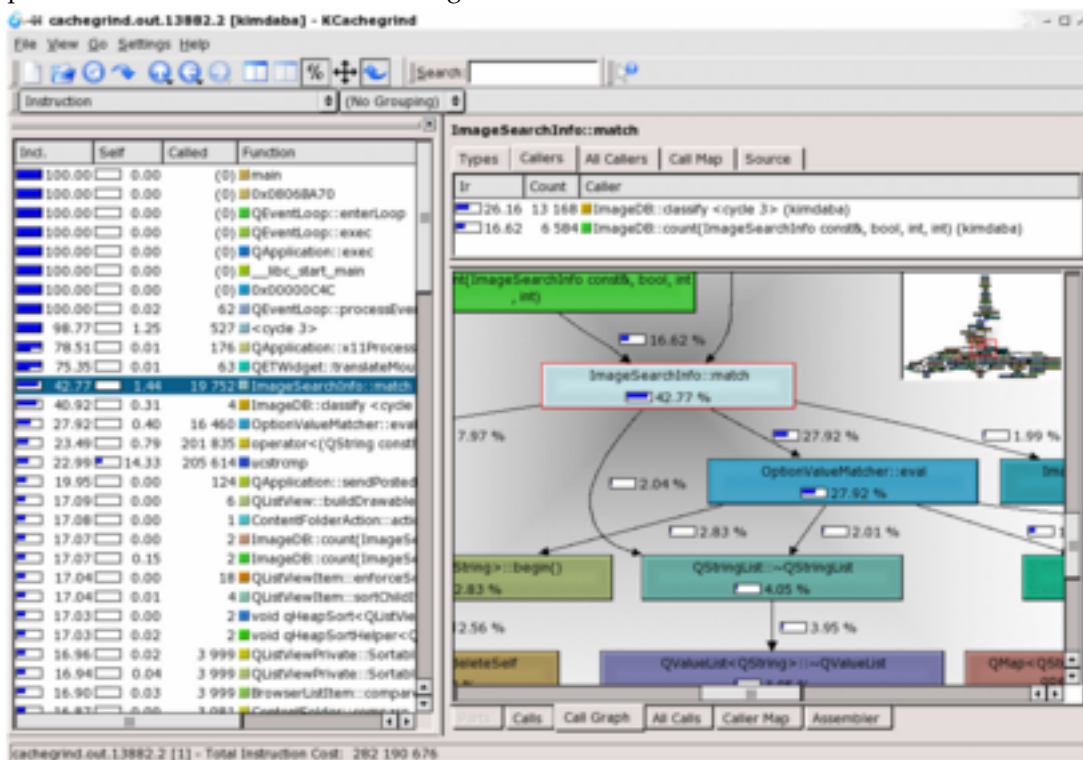
A word of caution when talking about profiling. Make sure to profile your application in a realistic context. Thus if you are profiling an email application which you believe is taking too much time to show an overview of the emails in a given folder, make sure that you profile with a folder with hundreds of emails (which is likely your normal situation) rather than with a folder with say five emails.

When profiling using unrealistic data sizes, the risk is that you will be fooled to believe that a wrong part of the application is what takes time - Imagine that an email could be in one of five states where an icon was used to indicate the state. Now imagine that each icon takes 100 msec to load from disk, while each email takes 50 msec to load. Profiling with 5 emails will shows that 500 msec is used in loading icons while 250 msec was used for loading email. This would indicate that you should spent time optimizing loading of icons. However had you profiled with 500 emails, still only 500 msec would have been spent on loading icons (there are only 5 different icons to load), while 25 seconds was used on loading the emails.

2.2.2 KCacheGrind

KCacheGrind is indeed an impressive application which make otherwise difficult data easy accessible. Try looking at how much data calltree dumps per second of program execution, and imagine you more or less should look at it using a normal editor (which to some extend is the case with tools like gprof - the open source market leader for profiling tools till calltree/KCacheGrind kicked it completely out of the market).

Below you may see a screen dump of KCacheGrind in action, and in the rest of this section an overview of the part of the KCacheGrind GUI will be given.



The KCacheGrind view is divided in two parts. On the left side you see the functions of the application - or rather top 100 with respect to resources used. On the right side you see one or two views each showing a number of tabs. The tabs can be moved around between the views, or even be hidden completely, if the tab do not show information of interest to you.

The following list described the content of each tab. Some of the information in the tabs are unfortunately unknown to the author and has therefore been skipped.

- **Calls** - Shows which function the current function calls.
- **Callers** - Shows which functions calls the current function. (If you think this is the same as `Calls`, then please read the text for the two again)
- **All Calls** - Shows which function the current function calls, plus which functions they call and so on.
- **All Callers** - Shows which functions calls the current one, plus which calls them and so on.
- **Call Map** - Shows the function which are called by the current function as a pyramid landscape, where the area of a pyramid is proportional to the amount of time spent. See snapshot below.
- **Caller Map** - Similar to `Call Map` but for function calling the current one, rather than being called by the current one.
- **source** - Shows the source of the current function intermixed with the time figures.
- **Call Graph** - This is the most important tab of them all - it shows the call graph graphically as can be seen in the screen shut above.



2.3 The massif tool

The last tool from the `valgrind` suite we will have a look at is `Massif`. `Massif` is at the same time very simple and very useful. `Massif` will generate a graph showing at which time in your application how much memory has been allocated, and from which part of the application the memory was allocated. Simply run your application like `valgrind --tool=massif`. The snapshot below shows you what a `massif` graph looks like.

