

# Evolution: Components optimisation

Cecilia González Álvarez  
<m3gumi@gmail.com>

31st August 2006

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Women's Summer Outreach Program 2006 . . . . .	3
1.2	Environment . . . . .	3
<b>2</b>	<b>Early profiling</b>	<b>5</b>
<b>3</b>	<b>Main profiling</b>	<b>6</b>
3.1	Automation . . . . .	6
3.2	Timing tests . . . . .	6
3.3	Sysprof hints . . . . .	7
<b>4</b>	<b>The problem</b>	<b>12</b>
4.1	Accelerators . . . . .	12
4.2	Studying internals . . . . .	12
4.3	The last decision . . . . .	13
<b>5</b>	<b>Results</b>	<b>14</b>

# 1 Introduction

## 1.1 Women's Summer Outreach Program 2006

In June, the GNOME people announced at their web page [5] a sort of internship like the Google's Summer of Code, but only for women. It consisted in developing a project during July and August chosen by the participants and related to GNOME. Moreover, the students would have the support of mentors from the GNOME community.

My mentor has been Federico Mena Quintero and I want to thank him for all the help he gave to me to develop my project.

The main goal of the project I chose was the optimisation of the switch of components in Evolution. Each of these components (mail, contacts, calendar, tasks, memos) took about 1 second to switch. This delay might cause user's dissatisfaction, so I expected to reduce the time of switching to 0.2 seconds.

The process of optimisation is composed by the following goals: first, accurate time measuring of the switch for each component; second, profiling of the code involved (where is the time being wasted?) to detect the priorities for optimising and finally, patch the code to improve the performance according to the results of profiling.

## 1.2 Environment

The project has been developed on a Pentium 4 (2 GHz), RAM 256 MB, with Linux 2.6.17, distribution Debian unstable.

### JHBuild

The first step in the project was the installation of the tool JHBuild [2] for building Evolution and its dependencies from CVS. The newest Evolution is then installed under my home without interfering the system installation. The common flags for the compilation of the modules were '-Wall -g -Os -fno-inline', because we need instrumentation of code for the profiling.

The module being compiled by JHBuild is obviously 'evolution'. All its dependencies are automatically checked-out, configured and compiled. As Evolution need a lot of modules to be completely built, the compilation takes some hours... During the configuration/building of many modules, JHBuild failed. The main reason was the lack of several development packages, (almost) immediately solved with apt-get, looking at the messages of the building or the file config.log. However, I racked my brain with some issues:

- I hadn't installed a devel package of x11. Pango didn't complain about it, but gtk+ didn't compile. Cairo complained about no X found, but continued the installation, so I didn't notice it firstly. Finally, I discovered it in the config.log of cairo. Checking the log I saw that the header Intrinsic.h haven't been imported, which is in the libxt-dev Debian package that I hadn't installed.

- The most difficult issue to detect was related to my shell paths. Some modules didn't link properly because they intended to link with the libraries in my system. The problem was in the `$PATH` and `$PKG_CONFIG_PATH` variables. In my `.bashrc` file I have defined paths like `'PATH_X = /path/1:/path/2'` instead of `'$PATH_X = $PATH_X:/path/1:/path/2'`. When the `jhbuild` environment starts, it takes firstly the definition of variables from the `.jhbuildrc` file (which keeps the paths of the custom installation), and then from the `.bashrc` file (standard system environment). So, with the wrong way, it got the path of the system firstly and then the path of the `jhbuild` installation instead of `jhbuild:system`.
- The latest version of library of my system was required (`gobject`), but the required version (2.12) was not available at Debian unstable. I got it from Debian experimental and no problem.
- With `hal`, I had problems with the version of `libvolume-id` too. The version required was older than the one available in Debian. Luckily, I found an RPM package of the old version and I installed it (after its deb-inisation).

#### **More tools**

- `Sysprof` [4]: the CPU profiler.
- Charts are made with `OpenOffice` and `Gnumeric`.

Mail	0,34	0,35	0,45	0,46	0,55
Tasks	0,15	0,14	0,18	0,24	0,25

Table 1: Some results of consecutive manual switches between components mail and tasks, from left (the first) to right (the last).

## 2 Early profiling

The firsts tests taken of Evolution were made to get used to sysprof and the actual behaviour of the switching of components.

Although the samples of sysprof were taken by manual switching, they could show some interesting information about the childs of the switch function that consumes the more.

However, timing tests can provide more accurate information to centre our profiling efforts later on a specific case. For example, we can start measuring the time for a whole switch in the function `e_shell_window_switch_to_component()` (`e-shell-window.c`), printing the elapsed time that we got through struct `tms` or a `Gtimer`. And with some manual switches between components mail and tasks we got some interesting results that we can see in table 1. The more switches we do, the more the time of switching increases, or at least it's what it seems at first glance. At this point, the best thing we may do is to automate the switching of components to get more samples.

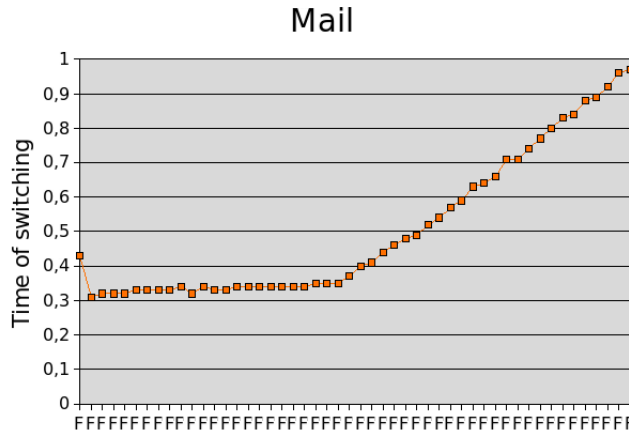


Figure 1: 100 consecutive switches of component mail.

### 3 Main profiling

#### 3.1 Automation

There are several ways to automate applications, like the frameworks LDTP [3] or Dogtail [1], but I chose to automate the switches modifying the code for some reasons: first, I had modified the code yet for getting times; second, it was the faster solution of automation and I was pushed for time. Of course, that isn't the best choice if we want change frequently the automation, which it is not this case. So, I added an extra button on the sidebar of Evolution which will switch the times that we indicate and between the components that we want to.

#### 3.2 Timing tests

Through the automation, we switch 100 times the component mail. The first switch takes 0,32 seconds and the last 1,34 seconds. It's clear that time increases for each new switch. We can see this increase more graphically in figure 1.

We got some more time samples of switching 100 times other components: contacts (figure 2), calendar (figure 3), tasks (figure 4) and memos (figure 5). As we can see in these charts, time doesn't seem to grow as it does when component mail is switched.

And what does happen if we switch between the five components? The chart of figure 6 represents 50 switches between mail, contacts, calendar, tasks and memos (in this order). It seems that the execution of the switch of mail affects to the execution of the other components, specially the component that switches after mail, contacts. In fact, if we switch all the components except for mail, we get the results of figure 7, very different from those we got with mail

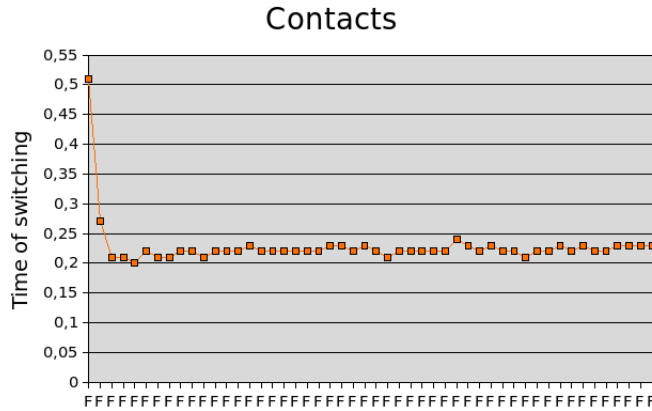


Figure 2: 100 consecutive switches of component contacts.

implied. Therefore, it means clearly that the main problem lied in the switching of component mail, and that it should affect other components.

Every switch between components the functions `component_view_deactivate()` and `component_view_activate()` are called, in order to deactivate the last component and activate the new component. I put timers to measure these functions and I got that for mail the time of activation is, firstly, over the time of deactivation and both times increase in every iteration. But after many iterations the time of deactivation increases over the time of activation. For example, the first switch takes 0,03 seconds to deactivate and 0,34 to activate. However, after 300 iterations we got 1,36 seconds to deactivate and 1,12 seconds to activate the component. Due to the growth of the time for deactivate mail I got in the chart of figure 6 the increase of component contacts, which should deactivate mail.

Besides, an informal memory test (with the information given by command `ps`) informed that the consumption of resident memory increases in every switch. That may indicate that we had a memory leaking problem.

### 3.3 Sysprof hints

It was time to take some samples with `sysprof`. If we run `sysprof` under the latest iterations we'll get more samples of the functions that are causing the increase of time. The most valuable information that we can get then is the function that takes the more to be executed, `em_folder_view_activate()`. This function calls a lot of methods. I measure the time of them to see their behaviour when we switch between components. The more significant results are showed in figure 8. As we can see, the function that take the more to be executed is `bonobo_ui_util_set_ui()` and `e_menu_activate()`, and both of them increase their time each iteration that they are executed.

Let's see what did `sysprof` say about the slowest function. In figure 9 the

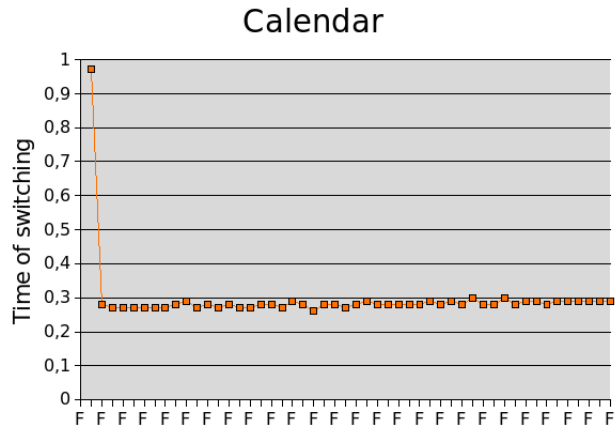


Figure 3: 100 consecutive switches of component calendar.

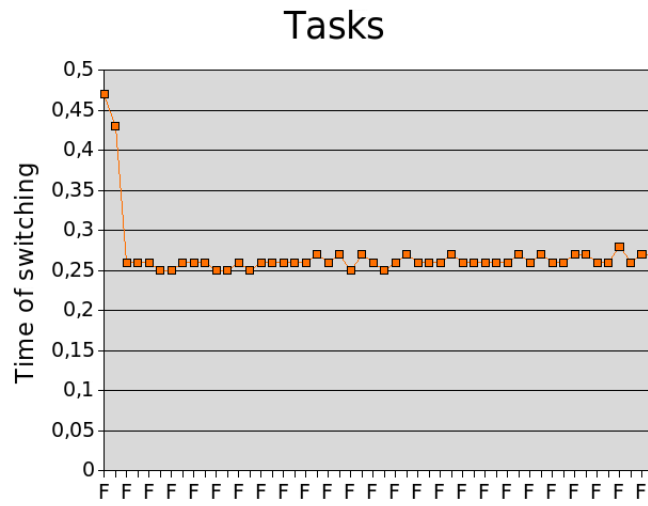


Figure 4: 100 consecutive switches of component tasks.

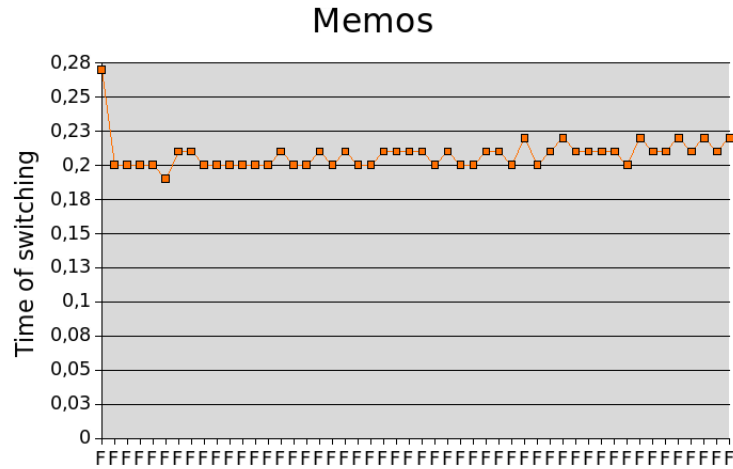


Figure 5: 100 consecutive switches of component memos.

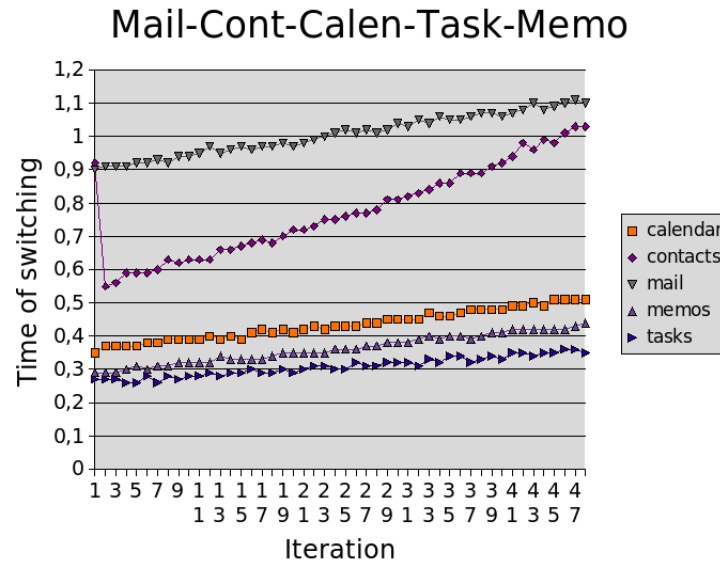


Figure 6: 50 consecutive switches of mail, contacts, calendar, tasks and memos (in this order).

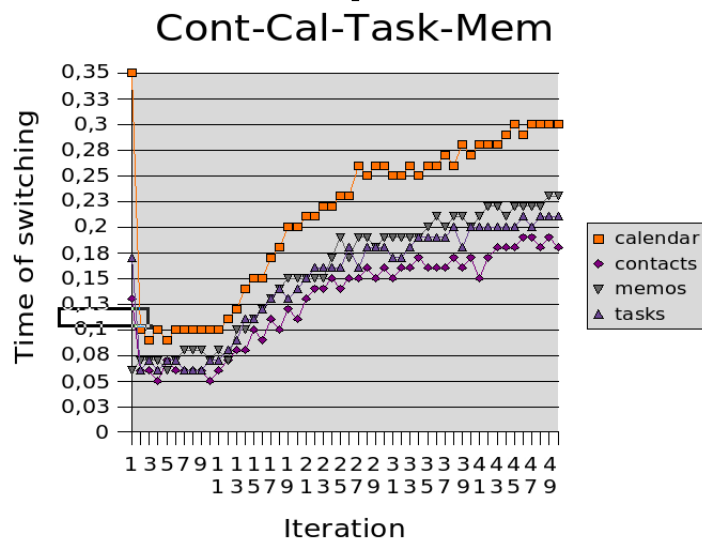


Figure 7: 50 consecutive switches of contacts, calendar, tasks and memos (in this order).

function can be seen spread when we get samples from switch number 0 to 50 and from 50 to 150. We reach the function `gtk_widget_add_accelerator()`; the first column of cumulative times (iteration 0 to 50) has a different value from the second one (iteration 50 to 150), which has a greater value. That indicates which was the bottleneck we were looking for!

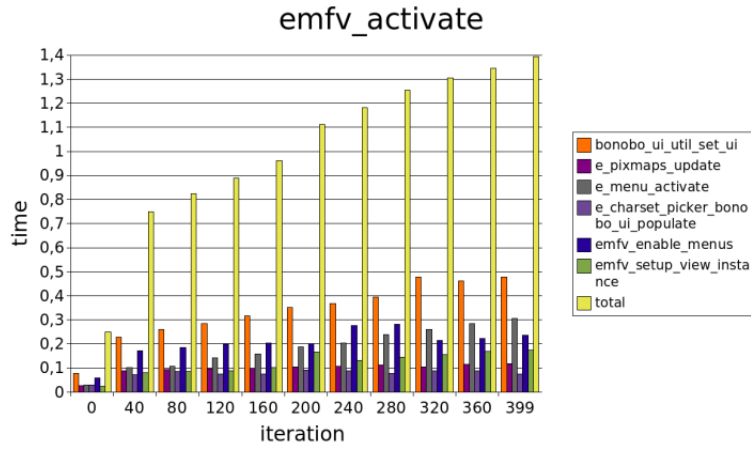


Figure 8: Growing of the function emfv\_activate and its childs.

Descendants	Self	Cumulative ▲	Self	Cumulative ▲
▼ emfv_activate	0,00	49,89	0,00	47,29
▼ bonobo_ui_util_set_ui	0,00	16,52	0,00	16,55
▼ bonobo_ui_component_set	0,00	16,50	0,00	16,55
▼ impl_xml_set	0,01	16,50	0,00	16,54
▼ Bonobo_UIContainer_setNode	0,00	16,49	0,00	16,54
▼ ORBit_c_stub_invoke	0,00	16,49	0,00	16,54
▼ _ORBIT_skel_small_Bonobo_UIContainer_setNode	0,00	16,49	0,00	16,54
▼ impl_Bonobo_UIContainer_setNode	0,00	16,49	0,00	16,54
▼ bonobo_ui_engine_xml_merge_tree	0,00	15,85	0,00	15,95
▼ bonobo_ui_engine_update	0,00	15,37	0,00	15,65
▼ do_sync	0,01	15,14	0,00	15,50
▼ bonobo_ui_engine_sync	0,03	14,91	0,04	15,28
▼ bonobo_ui_sync_state	0,01	9,02	0,00	10,33
▼ impl_bonobo_ui_sync_menu_state	0,01	5,30	0,01	6,97
▶ gtk_widget_add_accelerator	0,02	1,83	0,01	3,66
▶ In file /usr/lib/libgobject-2.0.so.0.1200.0	0,18	0,68	0,23	0,61

Figure 9: Left: % of time with samples of iteration 0 to 50 of consecutive switches. Right: % of time with samples of iteration 50 to 150 of consecutive switches.

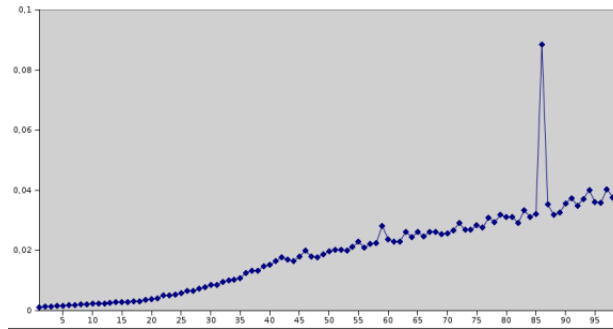


Figure 10: Behaviour of function `g_memmove()` invoked in `quick_accel_add()`.

## 4 The problem

### 4.1 Accelerators

From the conclusions of the previous section we may identify the problem somewhere in `bonobo-ui`, when the menu accelerators are added. The operation of adding an accelerator to a `gtk` widget drive us to the function `quick_accel_add()`. I put timers in every method invoked by this function, and I found out that the problem was caused by `g_memmove()`, as we can see in figure 10. Obviously, if a function that moves data gets slowly in each iteration it means that this data is growing... memory leak.

### 4.2 Studying internals

At this point we should find what is actually happening when accelerators are added to the menu widget. So, we print the information about the widget and accelerator which is added when the function `gtk_widget_add_accelerator()` is invoked in the synchronisation of the menu by Bonobo. Studying the output of many switches of component mail, we got that for each iteration we have 42 accelerators more which will remain in memory till Evolution is closed. Specifically, they are only six generic accelerators (`*Control**Shift*s`, `F1`, `*Control**Shift*w`, `F9`, `*Control*w`, `*Control*q`) but that are added seven times. Some different accelerators are added many times too, but their closures are removed. Therefore, we are losing time due to redundant closures added and due to the memory leak.

To know which is the cause of all that, we might go into the application internals through GDB. For example, we set a conditional breakpoint in the function that parses the accelerator, breaking the execution only when a particular accelerator is added. If we trace the stack after the break, we can see that the (far) origin of the call lies in different XML files (which may contain the description of the UI). However, the problem weren't the XML files, but the

merge of the XML tree which reads those files.

After exploring many debugging outputs (enabling for example `WIDGET_SYNC_DEBUG` or `XML_MERGE_DEBUG` in `bonbo-ui-engine.c`) and more GDB stack traces I realized that an accelerator was changed whenever the node of the XML tree that represents a specific menu widget was marked as "dirty". In the process of merging the several XML trees a node could be set "dirty" depending on the command that represent. Some commands need to be synchronized; this sync is transferred to the node, for example in `impl_bonobo_ui_sync_menu_state()`. But the problem that this function had was the chaotic addition of accelerators which made the expensive growth of structures that we had checked previously.

### 4.3 The last decision

As I didn't see a way to avoid that a node was set as "dirty", I thought that the best place to set the retaining wall was in the menu synchronising.

The patch I wrote is accessible at [http://m3gumi.iespana.es/patch\\_0828.diff](http://m3gumi.iespana.es/patch_0828.diff). It respects the fact that we can have a widget with multiple accelerators and that the same accelerator can be added many times in the same accelerator group. The code only avoid the accelerator addition if it has been added previously to the same widget (which is the common case).

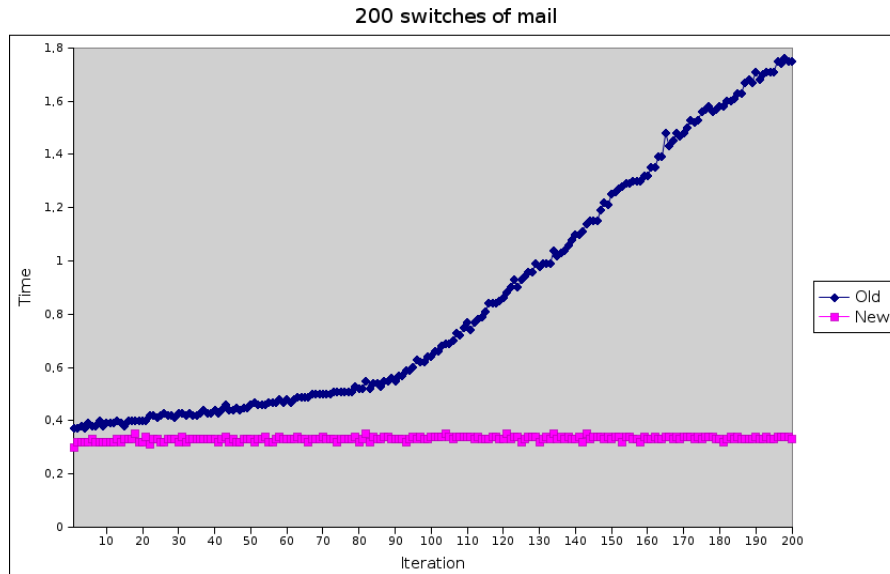


Figure 11: Behaviour of the old and new switching code.

## 5 Results

In figure 11 we can see a chart with the difference of mail switches without (blue) and with (pink) the patch. For those 200 iterations we got satisfactory results; time now keeps constant.

And what about the other components? Their time remain constant, and, the most important thing, it has diminished (around and under 0,1 seconds), as we can see in figure 12 (compare these results to those we got without the patch, in figure 6 or 7).

I made the chart of figure 12 with so much switches because want to remark that after a lot iterations the switch of mail becomes a little slower and this fact should be investigate.

