

Improving Performance

Documentation »
ACCESS Linux Platform Native Development »
Programming Guide »
Improving Performance

In order to achieve acceptable performance in our applications, we must always keep it in mind when making design decisions. It is often the case when developing applications, that the choice of design for implementing the UI will have the greatest overall impact on perceived performance.

Performance Tips ^TOP^

The following sections give some useful tips that may or may not apply to your application. Follow those that do, and you should see an improvement in your application's performance.

GTK/Glade UI ^TOP^

- Empty UI is shown before all of your data is ready (e.g. email application displays UI first, then fills in list of email messages)
- Populate empty UI in response to a callback from `alp_max_set_window_shown_callback()`.
- Put each top-level window definition into its own Glade file.
- Avoid reloading Glade files, especially large ones. Call `alp_bundle_acquire_glade_xml()` only once per file, and cache the result.
- Keep windows hidden until they are fully configured. Set the "visible" attribute in Glade to "false", then later call `gtk_window_show()` after initializing widgets.
- Don't connect callbacks to "changed" signals of widgets while data is being loaded.
- Load and render only as much data as fits on the screen. Consider use of `AlpProxyCacheModel` to do this automatically (see the API reference for more on `AlpProxyCacheModel`).
- Hide UI elements when modifying them (for instance, when populating controls such as list views and category combo boxes with data).
- Detach data stores when doing large UI updates

Notifications ^TOP^

- Avoid using `alp_notify_register_launch()`. Instead, whenever possible use `alp_notify_register()`.
- Call the `alp_notify_done()` function as soon as possible.

- Use the <notifications> tag in your application's Manifest.xml file to register for notifications whenever possible.
- Avoid registering for the "boot" notification (ALP_NOTIFY_EVENT_NOTIFY_BOOT). For just about all cases where you might want to do this, the "Install" notification should be sufficient (ALP_NOTIFY_EVENT_NOTIFY_REGISTER).

Global Settings ^TOP^

- Use a single global settings context for the life of your application. This avoids multiple calls to alp_settings_open() and alp_settings_close().
- Use transactions (see alp_settings_begin_transaction()) when reading or writing a large number of keys.
- Use alp_settings_set_multiple() to operate on multiple keys at once. This minimizes IPCs.
- Register only once for a group of related Global Settings notifications. For instance, if key-change notifications for keys "/a/b/c1" and "/a/b/c2" are needed, consider using

```
alp_notify_register("/alp/a/b")
```

SQL ^TOP^

- Use transactions (alp_sal_transaction_begin()) when performing multiple database operations together.
- Optimize SQL queries. See http://web.utk.edu/~jplyon/sqlite/SQLite_optimization_FAQ.html.

For more tips on using SQLite, see "Use of SQLite."

XML ^TOP^

- Use the simple XML parser "GMarkup" (<http://library.gnome.org/devel/glib/unstable/glib-Simple-XML-Subset-Parser.html>). If you need full XML, but don't need DOM, try libexpat (<http://expat.sourceforge.net/>).

Libraries ^TOP^

- Don't link to libraries you don't use. Check by using ldd.
- Use a version script for libraries you create to remove non-public symbols. Fewer exposed symbols mean less memory usage and faster application start-up.

Miscellaneous ^TOP^

- Consider "lazy evaluation." Delay loading and processing data until the user asks for it.
- Cache unchanging data. If you are doing work to display data, see if you can keep those results.

- See if expensive or wasteful algorithms can be removed or replaced (For instance, use AlpProxyCache instead of GTK's tree view, use quicksort instead of bubble sort, etc.).
- Explore using a different memory type. The slowest memory is NAND (generally accessible from "/"), followed by external memory (such as is found on an MMC card; this is accessed at "/mnt/<something>"), followed by RAM ("/tmp").

Use of SQLite ^TOP^

SQLite provides a powerful facility to maintain persistent data, and supports complex data relationships, but as with any database, you can't use it blindly and expect acceptable performance. For example, consider how SQLite makes use of temporary tables. Here are the interesting bits:

- By default, TEMP[ORARY] tables are put into a randomly named file in /var/tmp. According to the FHS specification, /var/tmp is supposed to be persistent and survive reboots. On an ACCESS Linux Platform device, this is true, and /var/tmp is in NAND. That means any operation which creates or uses a TEMPORARY table will involve file creations, locks, writes, and syncs to the NAND file system; which is very expensive.
- Creating the temporary file in /var/tmp involves opening /dev/urandom to retrieve some random bytes, presumably for seeding the random name of the temporary database file. Reading from /dev/urandom also involves a kernel dive, and can consume a small portion of the kernel entropy pool.

There are several ways you can improve SQLite performance:

- you can place the temporary database files somewhere besides /var/tmp, e.g. /tmp, which is an in-memory file system (see below). To do this, you need to use this SQL command:

```
PRAGMA temp_store_directory = '/tmp';
```

- you can store temporary tables directly in application private memory, without touching a file at all, by using this syntax:

```
PRAGMA temp_store = MEMORY;
```

- you can also ATTACH tables in different file-systems to an existing SQLite3 database connection. For example, the Bundle Manager database is primarily in a table on /var, with a single table in a file in /tmp, and a view in a memory temporary db. This provides distinct life-time characteristics: most of the tables are updated rarely, and need to survive reboot, so they go in NAND on /var. The live_metadata table is changed often and is shared between processes but doesn't need to survive reboot. The view doesn't need to be shared at all and can go into memory but each instance of the Bundle Manager library needs to create its own instance of the view, since it isn't shared.

Note:

- /tmp is on a temporary filesystem, which directly consumes RAM. This RAM is effectively unpagable (since we don't have swap), so anything on /tmp is taking up RAM that isn't available for other processes to use, and cannot be charged directly to the owning process.

- 'MEMORY' databases are not stored on a filesystem, they are private to a process and will be appropriately charged to that process. (Like all RAM consumption on our system this is also unpagable, but at least it's clear who is taking up space.)

Data Window ^TOP^

There is an alternate approach to populating the entire list that is currently being investigated. It is to only populate the ListStore with the only number of items displayed on the screen. When the user scrolls, the "window" on the database is updated to contain the new set of items to be displayed.

The "data window" approach is looking very promising as a means to achieve good performance when displaying a subset of what might be a very large data set. We are also looking at the possibility of refactoring the current implementation to create a shared library that implements the generic aspects of a data window. This library could then be used in place of the standard Gtk+ GtkTreeView

{moscomment}