Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

The World of OpenOffice

Michael Meeks
Novell, Inc.
`mmeeks@novell.com`

Abstract

In this talk I will present some of the issues facing OpenOffice.org, particularly related to: performance, interoperability, buildability, ABI / engineering and release practice. We'll look at how to build the beast, the UNO component model, and iterate a quick hack before your eyes. We'll also show some of the flash new features including the Gnome desktop integration work.

1 A friendly giant

The OpenOffice.org source base is one of the largest monolithic Free software projects in existence, even with the pre-compiled mozilla binaries for several architectures stripped out:

Project	Source bz2 (MB)
Mozilla 1.4.1	31
Linux 2.6.7	33
GNOME 2.6.2	108
OO.o 1.1.2	160

OpenOffice.org (OO.o) represents one of the largest single contributions to Free software ever. Given this, it is somewhat incredible that Sun immediately settled on a licensing scheme in that is both liberal and substantially symmetric.

OpenOffice.org is licensed under two licenses:

- `LGPL` – the familiar, and best Lesser GPL.
- `SISSL` – essentially X11 with trip-wires for malicious UNO API, and XML file format compatibility breakage.

While it is necessary to share copyright with Sun by signing the Joint Copyright Assignment (JCA)[2], the use of OO.o code in StarOffice can be considered as being achieved under the SISSL[3] provisions.

Thus there is clearly huge potential for add-ins, integration with proprietary data-feeds, macros, etc.

2 Sun's dilemma

Sun's StarOffice product substantially consists of the OpenOffice.org core, as seen in public CVS, with the addition of a few extra proprietary modules. While this means that all the latest bug fixes are available in public CVS, it creates a number of frustrating artificial problems:

2.1 Release Engineering

- `minor release cycles` – there is a correct separation of commercial updates; of around once per quarter; thus this tends to be the frequency of minor OO.o releases regardless of bugginess.
- `release patch-size` – there is a fixed upper-bound on the size of a cus-

tomer patch download, thus ABI alterations in low-level libraries which would have a large knock-on effect, are forbidden.

- `ultra conservatism` – since customer updates are infrequent there is little incentive to back-port fixes to the stable branch; so many, trivial but high-impact fixes don't make it.
- `major release cycles` – for reasons unknown StarOffice works on an 18-month release cycle, so—at times (given freezes, etc.), it is possible to punt a feature / fix by nearly 2 years.

Clearly many of these problems make the OO.o development process somewhat cumbersome.

2.2 Portability Engineering

In contrast to many Free software project, StarOffice and hence OO.o, is designed to run on a broad spectrum of operating systems and versions. By contrast, e.g. GNOME applications, would typically require the latest version of GNOME to run.

This creates a number of interesting, hard-core engineering issues, and shows up the true state of Linux as a robust platform for ISVs.

For example, for font discovery much Linux software will link to the pleasant fontconfig library, and use purely client-side font rendering. OO.o in contrast has to run on older (or newer) platforms where there is either no fontconfig install, or it has a changed ABI, or it is badly configured. Thus the OO.o font discovery method uses the following heuristics:

- `fontconfig` – since this may not be available, we try to `dlopen` it, hook out various symbols, and extract a simple list of font filenames.
- `chkfontpath` – Red Hat, and others once shipped this tool which dumps a list of font paths; we try to `popen` and parse the output.
- `hard-coded paths` – various directories such as `/usr/X11R6/lib/X11/fonts/truetype` are known to be a good bet, and are scanned for fonts, including several language specific variants.
- `X server query` – the X server is queried to see what it can do, and a load of XLFDs are parsed.
- `internal fonts` – whatever internal fonts, and font-metric files we distribute are added to the mix.

Naturally, after doing all this work, we build a OO.o specific cache of much of the information, to accelerate subsequent startup.

This heavily engineered approach is not constrained to any one API-set, or technology—so, e.g., OO.o will attempt to use either lpr or cups for printing in a dynamic fashion.

Even glibc problems show up in Figure 1.

In addition, the cross-platform nature of OO.o and the unpredictability of the Linux feature-set (particularly the C++ ABI), leads to a large number of software packages being included inside the OO.o build itself. Thus, a stock OOo would include it's own compiles of (at least): `python`, `freetype`, `zlib`, `expat`, `libdb`, `NAS`, `neon`, `curl`, `sane`, `myspell`, `Xrender`.

As is probably obvious, this level of old platform support, and dependency aversion is hard to get enthusiastic about.

```
typedef struct {
    struct { long status; int spinlock; } sem_lock;
    int sem_value;
    void *sem_waiting;
} glibc_21_sem_t;
    /*
     * XXX this a hack of course. since sizeof(sem_t) changed
     * from glibc-2.0.7 to glibc-2.1.x, we have to allocate the
     * larger of both XXX
     */
#ifdef LINUX
    if (sizeof(glibc_21_sem_t) > sizeof(sem_t))
        Semaphore = malloc(sizeof(glibc_21_sem_t));
    else
#endif
        Semaphore = malloc(sizeof(sem_t));
}
```

Figure 1: compatibility with old `glibc` versions

3 Community Issues

In addition to these unusual constraints, the OO.o project is encumbered by acute tooling and collaboration inadequacies.

Perhaps the most serious problem, is that it appears CVS was not designed with 200+ MB of source / binaries in mind. Thus, even basic operations, such as a `cvs tag` can take up to a couple of hours, and are frequently blocked by robots slowly traversing the repository.

Secondarily, the collab.net SourceCast system adds a level of bureaucracy, and lack of responsiveness which when combined with being totally un-fixable makes for an unnecessarily painful experience. It seems likely that SourceCast is ideal for the use of existing, established Free software projects, or even newly formed projects—but it stumbles with OO.o. Furthermore, using closed software for Open Source collaboration is an intrinsically interesting decision.

4 The other side of the coin

4.1 `http://ooo.ximian.com/`

To make up for the existing inadequate webtools, and documentation we provide several ‘external’ tools of interest.

- `hackers guide` – a Linux focused, hackers guide on how to build, iterate, and some basics of the OO.o code structure.
- `LXR/Bonsai` – basic web tools without which navigating the OO.o source is substantially more difficult.
- `bug filing` – a gateway that demangles the curious user-focused issue filing process, and allows bug filing directly against given code modules.
- `Planet OO.o` – the obligatory RSS aggregator.

4.2 ooo-build

The process of productising OO.o into a Linux package is filled with pain; so to amortise this a collaboration has coalesced between various Linux vendors: Novell ne Ximian, Debian, Red Hat, SuSE, Ark, and PLD Linux around *ooo-build*.

ooo-build provides a growing set of useful patches many of which may arrive in OO.o in many months time; indeed all our work is intended to go up-stream into OO.o. We also provide a simple patch sub-setting system, to allow vendors to select a suitable set of patches.

Many of the features associated with ooo-build are desktop integration, system integration, and GUI cleanup pieces; e.g.:

- attractive new icons
- native-widget rendering
- GNOME-VFS integration
- ergonomic & aesthetic fixes
- system library usage

The ooo-build wrapper is also intended to make OO.o substantially easier to compile with a familiar `./configure; ./download; make; make install` process.

5 Performance

Performance is an area ripe for substantial improvement in OO.o, however, poor performance is caused by many factors, and identifying the most important of these is not always easy.

5.1 Linking

The linker has a very hard time linking OO.o, and while this can be reduced by pre-linking, the architecture of OO.o—whereby the majority of the code is in shared libraries required not by the main binary—but by other shared component libraries, linked at run-time.

Ulrich's analysis of OO.o [1] shows that 20,000 relocations are performed during startup, which combined with lookups across multiple libraries gives 1,700,000 string comparisons to startup. The sheer size of the symbol tables and the lack of locality of reference in the linking process causes much of this work to fall outside the processors' cache—giving abnormally poor performance.

5.2 C++ issues

Some features of C++ exacerbate the problems of large symbol tables, and poor startup performance. The stripping / re-working of static initialisers has helped accelerate performance—these being replaced with a thread-safe late instantiation based on accessor method local static variables.

C++ is a very symbol-hungry language—particularly with respect to virtual functions, which create an unnecessary burden (Figure 2). Virtual functions, despite resolving to a simple function pointer export a symbol, which is referred to directly to chain to parent implementations. While of course this can often be resolved away at link time, in a cross-library situation it would perhaps be more efficient to dereference a parent vtable function pointer.

Similarly, since in theory at least, a single class can be implemented across multiple shared objects, even 'private:' methods export symbols.

In addition to these problems, a more proactive approach to pruning old, and redundant code has been adopted in the development branch, to reduce code footprint, and symbol count.

```
class Foo : public Baa {
    virtual void VFunc();
  private:
    void ExportsSymbol();
};
...
void VFunc()
{
    ...
    Baa::VFunc();
}
```

Figure 2: C++ virtual functions

5.3 Binary filter code

To shrink the OO.o footprint, a large chunk of creaking binary format code has been extracted, along with compatible chunks of the core. This code pre-dating the XML file formats scattered the process of serialisation across the code, and resulted in a complex, hard-to-maintain and increasingly irrelevant maintenance problem. In OO.o 2.0 it will be used only on the rare occasions it is necessary as a binary to XML filter.

5.4 system libraries

Shrinking the large number of internal libraries, on Linux systems, and increasing the number of libraries shared with the system is an important part of performance improvement in 2.0. It clearly makes little sense to have an internal gtk+ library when the system version is ABI compatible, and better maintained.

Using system libraries—e.g., neon—also reduces the pain of handling security updates in the built-in libraries.

5.5 mmap performance

Possibly the most significant speedup in the 1.0 to 1.1 transition was the process of forcing as much of the OO.o code into memory before attempting to run it. This gave a very noticeable win; this was implemented in a simple fashion with mmap, and a loop reading a byte from each page. Ideally of course, the underling operating system would be able to do better here.

6 Interoperability

In a world where a tiny fraction of people are using Free software, the ability to share documents in a loss-less fashion with other people is crucial to the adoption of OO.o.

Much work has been done in this area for 2.0, of particular note the row-limit in calc has been raised to that of Excel, and much work has been done on form controls.

There are also exciting developments in VBA interoperability. OO.o provides a VBA-like language: StarBasic, and by devious means it has been possible to extract VBA text from Office files for some while. Office for performance reasons however stores VBA in 3 forms: an SRP stream, a compiled form, and a compressed text form. Since these are authoritative in that order (the text providing only a final fallback), it was thought that effective export would entail reverse engineering at least the the compiled form.

However in recent time, yet more devious means have been discovered to export macros as text to Excel and have them run transparently to the user. This it turns out is the foundation of macro interoperability between Office versions 97 through XP. Thus work is ongoing to improve the macro support so crucial for effective Excel interoperability.

7 Desktop integration

Much of the work of ooo-build has been adopted in one form or another up-stream for 2.0, giving the prospect of a highly desktop integrated OO.o experience out-of-the-box.

To achieve this, the lowest levels of OO.o's cross-platform abstraction: the Visual Class Library (VCL) have been virtualised, and now the main-loop, and top-level windows on a GNOME system are handled by the gtk+ toolkit. In order to avoid a complete re-write of the widget system—we use a simplified theming system that virtualises only the rendering of widgets, allowing basic widgets to match the look of the rest of the desktop.

Similarly main-loop integration makes things such as integrating the gtk+ file-selector and other GNOME dialogs fairly simple. The main-loop integration was made substantially more painful by the mis-match between the recursive OO.o toolkit lock, and the non-recursive gtk+ lock. In order to reconcile these and provide a single, comprehensible locking pattern—after considerable thought we added hooks to gtk+ to allow a shared (recursive) lock to be used. This makes gtk+ use in OO.o virtually seamless.

8 UNO component model

OO.o provides a rich, and well documented component model, which is exported for the use of language bindings. The power of this, and its flexibility have resulted in active bindings for StarBasic, Java, and Python.

The UNO model is particularly interesting, since it consumes little overhead beyond a stock C++ virtual function call. In addition each class has associated, small compiled IDL type information. This can be used, to dynamically (at run time) construct bridges to other languages, and allow dynamic method invocation. While this adds a compiler version / ABI dependency to the OO.o core, it avoids the problem of creating stub / skeleton code which ended up consuming many MB before the dynamic approach was adopted.

9 Conclusions

OO.o provides an unusualm and particularly pathalogical case of a gigantic C++ project. This leads us to push the boundaries of the system, showing up several areas for potential improvement.

The ooo-build infrastructure provides a solid base for contributing work to OO.o in a familiar and accessible manner, and seeing the deployed results of your work quickly.

OpenOffice 2.0 will give substantial performance, code-cleanliness and interoperability improvements, in addition to many new features.

References

[1] Ulrich Drepper. How to write shared libraries, 2004. `http://people.redhat.com/drepper/dsohowto.pdf`.

[2] Sun Microsystems, Inc. Joint copyright assignment. `http://www.openoffice.org/licenses/jca.pdf`.

[3] Sun Microsystems, Inc. Sun industry standards source license. `http://www.openoffice.org/licenses/sissl_license.html`.