Sponsored by AMD

www.novell.com

# Investigate Footprint Shrinkage of OpenOffice.org

Michael Meeks <michael.meeks@novell.com>,
Tor Lillqvist <tlillqvist@novell.com>

## Overview

OpenOffice.org is mythically resource intensive, and as the largest free C++ application out there stress tests everything it touches: compiler, toolchain, I/O & swap algorithms, memory allocators and so on.

This document analyzes the footprint of the writer component (since this is typical), running under a GNOME desktop, though the results should be similar to those of other environments. All analysis was performed (unless otherwise specified) with an OO.o 2.0.4 build on a SLED10 system.

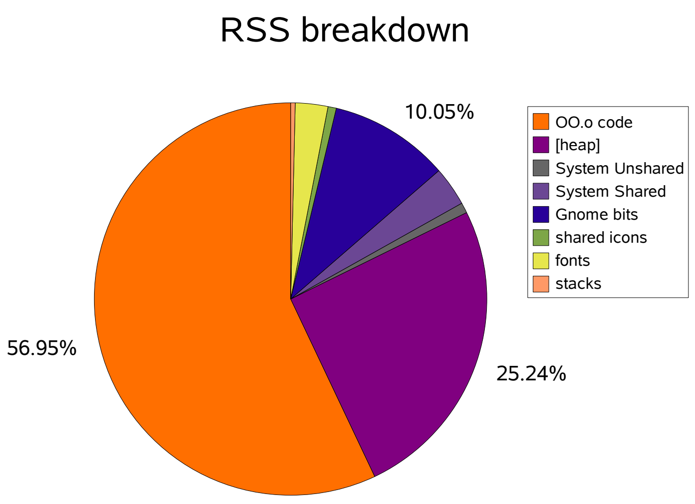
## Memory breakdown

### Overall pmap analysis

pmap is a standard tool for displaying information about process memory mappings. It can be used for simple, but reliable  measure of the memory problems:

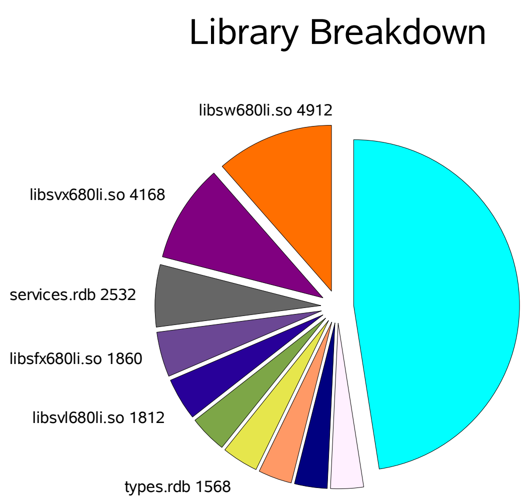
| | **Size / Mb** | **RSS / Mb** | **Dirty / Mb** |
|---|---|---|---|
| **Total** | 266 (104 unshared) | 75 | 25 |

So – we require 75Mb of physical memory to do no more than start up writer, and enter a few characters. Of this, only 19Mb (25%) is heap space. Thus to reduce memory consumption it will be necessary to reduce both heap and code sizes.

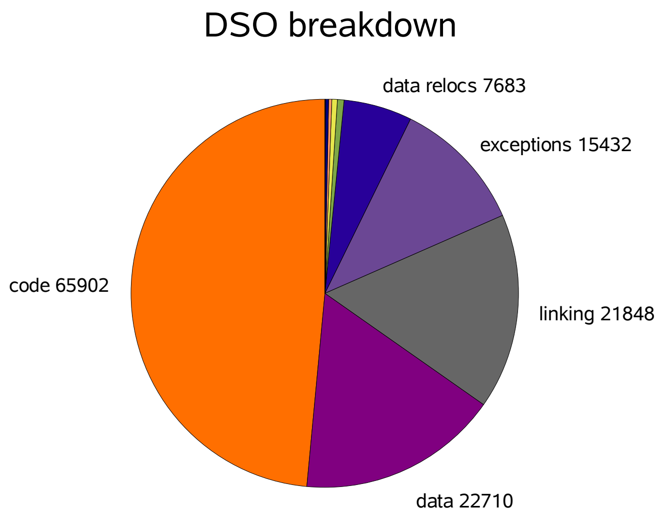


### Code breakdown analysis

We have ~43Mb of core OO.o code (RSS), from ~83Mb (Size). The complete OO.o code size is 125Mb, so clearly a combination of on-demand loading, and the component model separation effects a reduction of a factor of ~3 in code size. Unfortunately, from a per-module level analysis of the remaining code, it is clear that there is no single dominant library susceptible to removal.



Similarly analysis of overall section sizes using the Novell-developed relocstat tool shows that things appear fairly normal although the data (`.data / .rodata`) appears large, and linking (`.rel.plt, .plt, .dynsym, .dynstr, .got.plt, .got, .hash`) seems unfortunately large at 16% of total size. Code (`.Text`) is still the largest chunk; we compile with -Os however it is clear that in this area more compiler work to optimize more aggressively for size may be helpful.



***C++ / g++ optimization***

There are however several sub-optimalities in the C++ compiler, mainly the excessive emission of vtable related relocations. Cleaning this up yields both space and time wins, particularly startup wins – since all vtable relocations must be processed before executing code in a given DSO (Dynamic Shared Object). A complete solution requires 2 schemes

1. DSO initializers to do simple bitmap informed copies of vtables from their parents. This turns the painfully slow named relocation symbol lookup process into a single parent vtable lookup per DSO, per sub-class. This also turns a 16byte relocation (+ symbol) into a 2bit entity. **- 8 days**
2. vtable instead of PLT (Procedure Linkage Table) invocation. When implementing virtual methods, chaining to the parent implementation is a very common occurrence. If instead of a direct by-symbol call via the PLT we do an indirect call

via a parent vtable pointer, assuming that all calling code uses this convention we can save more symbols, PLT trampolines & GOT slots. **- 8 days**

The total saving here across the 125Mb of code is around 9Mb, ie. ~7%, so perhaps we can save 5Mb from the total size.

### *Extraneous libraries*

A fair number of libraries are not really required on startup – re-factoring could be used to remove these from the stack eg.

| **Name** | **Size / Kb** |
|---|---|
| sndfile | 344 |
| jvmaccessgcc3 | 160 |
| jvmfwk | 160 |
| avmedia | 144 |
| portaudio | 28 |
| **Total** | **836** |

While removing these libraries may marginally improve startup performance, the net memory gain is not substantial, for a substantial re-factoring cost. - **5 days**

### Component databases (.rdb) files

The .rdb files 10Mb total size, (4Mb RSS) are highly inefficient using an obsolete legacy format. When gzipped, these files shrink to 1Mb, so it should be easy to save >50% of the memory consumed here, yielding another 5Mb from the total size. There are however some API issues to overcome here relating to up-stream's ABI stability commitment – **5 days**

### Images.zip shrink

The initial design of accessible icons was not theme based, and included a single theme of accessible variants of ~all artwork into the same .zip file. Thus a certain amount of resources are consumed, by accommodating these. While only 26% of the icon size is accessible variants, this accounts for 35%+ of the directory size – which is permanently loaded. Thus we can save perhaps 200k moving to a theme based approach. - 5 **days**

### Heap Allocation

Profiling the heap allocation reveals a number of interesting issues of varying complexity.

| **Area** | **Size / Kb** | **Saving / Kb** | **Description** |
|---|---|---|---|
| hunspell | 3000 | 1500 | The dictionaries are loaded, and strings duplicated into memory instead of mmapping the dictionary files (and complicating the checking code) – **5 days** |
| OUString | 2000 | 1500 | The majority of strings are duplicates, and the UCS-2 representation also burns big chunks of memory, since almost all strings are programmatic and hence ASCII anyway. - **2 days** |

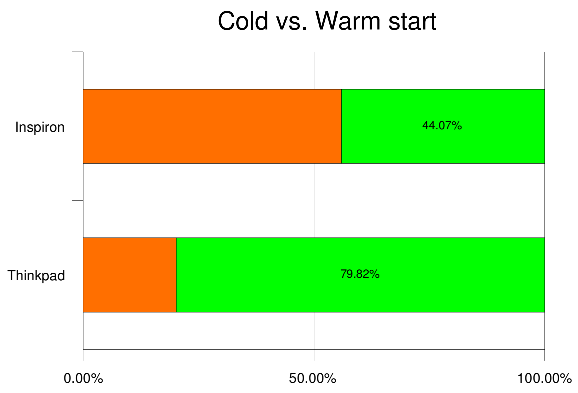
| | | | |
|---|---|---|---|
| configmgr | 3443 | 1000 | SimpleCheckingHeapMgr – 1.3Mb, stl node allocation – 1.2Mb, strings, obj references etc. consume the rest. **- 5 days** |
| SfxNewHdl | 512 | 512 | This is a buffer against OOM (to be freed in that case), but is (apparently) never used. **- 0.1 days** |
| 'package' zip code | 1022 | 500 | reading the large images.zip file creates a huge hash table with lots of duplicated string stems – **3 days** |
| component registry | 890 | 300 | The extensive use of OMultiTypeInterfaceContainerHelper which involves an inappropriate use of the (badly behaved for small sets) STL::hashtable burns the memory here. Also OStorePageData consumes a chunk of memory. - **1 day** |

String duplication is a rampant problem, analysis of string usage suggests that 90% (by size) of strings post startup are duplicates of other strings. The base string classes are immutable, so implementing a global unique string hash may be a simple and worthwhile solution here. It is notable too that malloc overhead is significant – a malloc based profiler registers 16Mb of allocations, where pmap reports 19Mb.

## Startup performance issues

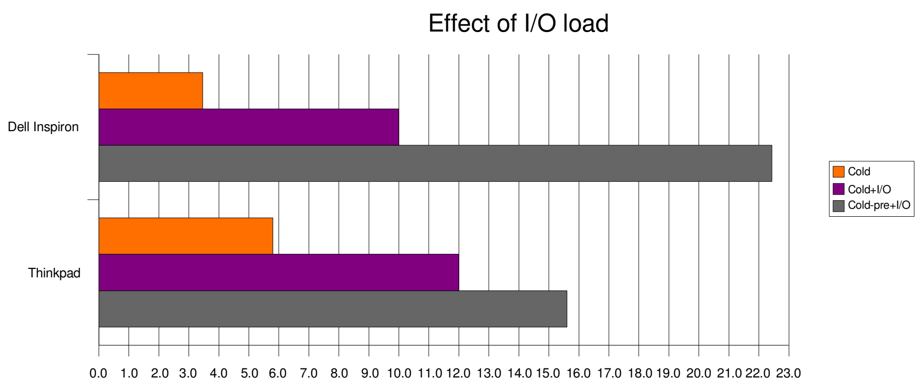
### Dominating Cold Start problems

The difference between cold and warm start times is substantial on typical hardware:



CPU time on the left and I/O time on the right as a proportion of total cold start up time. The  Inspiron is far older hardware, so the CPU costs of linking / startup dominate the I/O issues, but the Thinkpad is a far more recent CPU with larger cache.

A further problem is with small memory footprint situations – Linux I/O degrades very substantially in the presence of multiple I/O requests:



This shows how simulated I/O load in the form of seeks (in this case an ls -R /usr in parallel) causes dramatic degradation in cold start time. Unfortunately, low memory situations exacerbate this problem by necessitating swap usage to free up physical memory to read in large chunks of code, as can be seen this can easily turn cold start time from 3 seconds to 23 seconds. Pre-loading can help to reduce this problem.

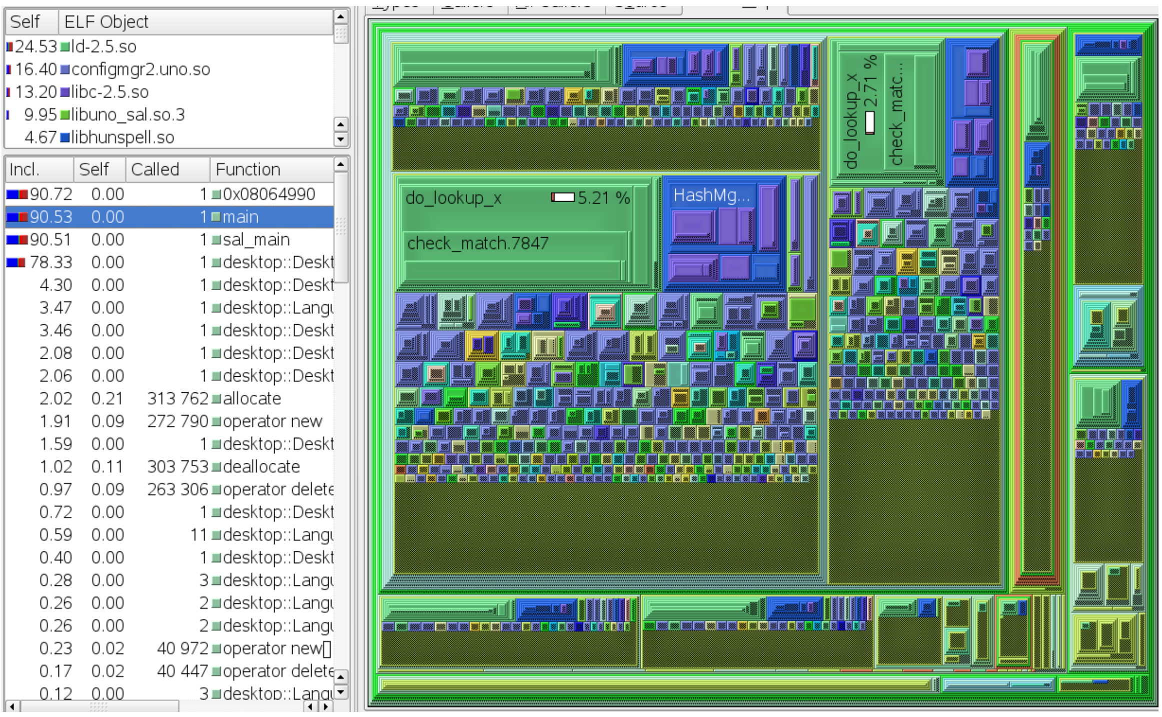
Ideally kernel work is required to quantify and address swap performance, I/O handling etc. Some substantial work has been done to reduce the amount of I/O performed at startup, but this work tends towards obfuscating on-disk structures in unpleasant ways. At root this is a kernel level issue.

Unfortunately, reliable cold start profiling is **extremely** difficult to repeat, or generate any reliable data from. This is particularly so due to the eclectic mix of syscall I/O and mmapped I/O. More research is required here – implementing a valgrind skin to generate memory (and mmap) traces, combined with syscall I/O traces would be a good first start. - **4 days**

### Cachegrind profiling

Valgrind is a program for debugging and profiling Linux executables using a synthetic CPU in software. Of course, its pseudo-CPU model is (at some level) extremely contrived, however it can give us some rough insight into the proportion of time spent in various areas. This simulation was tweaked for a processor with a very small cache:

```
valgrind --tool=callgrind --simulate-cache=yes --dump-
instr=yes \
       --I1=65536,16,32 --D1=65536,16,32 \
       --L2=131072,4,32 ./soffice.bin -writer
```



| Self | ELF Object |
|---|---|
| 24.53 | ld-2.5.so |
| 16.40 | configmgr2.uno.so |
| 13.20 | libc-2.5.so |
| 9.95 | libuno_sal.so.3 |
| 4.67 | libhunspell.so |

| Incl. | Self | Called | Function |
|---|---|---|---|
| 90.72 | 0.00 | 1 | 0x08064990 |
| 90.53 | 0.00 | 1 | main |
| 90.51 | 0.00 | 1 | sal_main |
| 78.33 | 0.00 | 1 | desktop::Deskt |
| 4.30 | 0.00 | 1 | desktop::Deskt |
| 3.47 | 0.00 | 1 | desktop::Langu |
| 3.46 | 0.00 | 1 | desktop::Deskt |
| 2.08 | 0.00 | 1 | desktop::Deskt |
| 2.06 | 0.00 | 1 | desktop::Deskt |
| 2.02 | 0.21 | 313 762 | allocate |
| 1.91 | 0.09 | 272 790 | operator new |
| 1.59 | 0.00 | 1 | desktop::Deskt |
| 1.02 | 0.11 | 303 753 | deallocate |
| 0.97 | 0.09 | 263 306 | operator delete |
| 0.72 | 0.00 | 1 | desktop::Deskt |
| 0.59 | 0.00 | 11 | desktop::Langu |
| 0.40 | 0.00 | 1 | desktop::Deskt |
| 0.28 | 0.00 | 3 | desktop::Langu |
| 0.26 | 0.00 | 2 | desktop::Langu |
| 0.26 | 0.00 | 2 | desktop::Langu |
| 0.23 | 0.02 | 40 972 | operator new[] |
| 0.17 | 0.02 | 40 447 | operator delete |
| 0.12 | 0.00 | 3 | desktop::Langu |

***Linking***

This is an area we have invested in substantially in the past. The above chart shows a SL10.2 system, with the -Bdirect and –hash-style=gnu optimisations present. These give a substantial improvement for machines with small L2 caches. However, since our -dynsort work went into binutils (as the new –hash-style=gnu option) it gained a pre-bloom filter. This is unnecessary and counter-productive for -Bdirect linking (it adds an extra cache miss per symbol), because we know which library contains the target symbol. More details are available at Optimizing Linker Load Times.

This work of course has a positive impact on all linking on the system, that is particularly helpful to systems with small caches. Extra work required here would involve:

- **binutils** – external relocation sorting by target library, and bucket library offset – 2 **days**
- **glibc** – re-factoring to improve efficiency, and to bypass the bloom filter in –hash-style=gnu for -Bdirect lookups. - 3 **days**
- **binaries –** due to implementation quirks -Bdirect doesn't generate sections for binaries – adding this would improve the linking of lots of small apps.

***Lazy Loading***

The -Bdirect linking implementation makes possible another linker optimization: Lazy Loading. (cf. MS' /delayload feature). Since the direct linking information lets us know where each symbol is referenced, in the case that we know that a library exports no Vague symbols that cannot be directly bound – it should be possible to defer loading this library until those symbols are referenced, detecting which library should be demand loaded from the direct linking data. Unfortunately this doesn't work well for libraries requiring early initialization, which includes a lot of our dependencies.

A number of binutils & glibc changes would be required to implement this, but it may be generally useful for all desktop apps – **15 days.**

### Config Manager

After linking, it is clear that configmgr is the next worst performing piece of OO.o. Indeed, previous measurements with -Bdirect linkage show it dominating startup, as it appears to on Win32. A host of inefficiencies relating to a previous architectural design (never fully realized) built towards a shared memory architecture for cross-process coherent config sharing. Unfortunately the level of inefficiency this imposes is incredible – it requires 79 lock/unlock pairs to fetch a -single- config key (eg.).

Re-factoring is ongoing here to make the code more readable, and substantial wins, perhaps a doubling of the configmgr performance should be possible here – **5 days.**

### Gnome VFS integration

This is used on both desktops to provide samba share access amongst other things. Unfortunately it is loaded on startup from ucpgvfs1.uno.so. which is the GnomeVFS "UCP" (Universal Content Provider) component. In normal use, the code in ucpgvfs1 is not needed at all, but it still has to be loaded to avoid a problem if it is loaded later through the "Proxy" UCP. On load
ucpgvfs1 pulls in libgnomevfs and many dependencies, otherwise unneeded during startup. Initializing gnome-vfs also reads and parses number of (potentially) scattered config files, some of which are fairly large.

Possible solutions here are to dlopen libgnomevfs and initialize it idly, or re-factor the ORB such that the proxy UCP problems go away, a prototype re-factoring in patch form exists – **3 days**

## Prioritized tasks

### Memory footprint reduction:

Prioritized by win/day

| **Task** | **Size / Kb** | **Duration / days** | **Saving Kb/day** |
|---|---|---|---|
| heap: SfxNewHdl | 512 | 0.1 | 5 120 |
| re-working rdb files | 5000 | 3 | 1,667 |
| heap: OUString | 1500 | 2 | 750 |
| *g++: Parent vtable chain* | *4500* | *8* | *563* |
| *g++: DSO initializers* | *4500* | *8* | *563* |
| heap: component registry | 300 | 1 | 300 |
| heap: hunspell | 1500 | 5 | 300 |
| heap: configmgr | 1000 | 5 | 200 |
| removing unneeded libs | 836 | 5 | 167 |
| heap: package | 500 | 3 | 167 |
| images.zip shrink | 200 | 5 | 40 |

The g++ work is somewhat complex and risky, unless done by the toolchain team, thus spending time several other tasks perhaps makes more sense.

### Performance improvements

Since the performance wins are rather harder to estimate here, this prioritisation is based on some considerable guesstimation:

| Task | Duration /days |
|---|---|
| configmgr | 5 |
| link: binutils | 2 |
| link: glibc improvements | 3 |
| VFS re-work | 3 |
| cold start valgrind / research | 4 |
| Lazy loading | 15 |

The configmgr is prioritized over linking, since SL10.2 improves the linking situation substantially across the system. Also the configmr is a major space hog as well, thus there is some overlap between the time & space savings here. However – the linking work does require re-visiting and further refinement.

## Conclusion

There is a lot to fix. If we split the time: 13 days of size reduction – which should yield a 2Mb heap shrink (12%), a 4% code shrink, and 5Mb from the .rdb files (4%). That would leave 7 days in which to tackle the configmr, and further improve the linking situation.