BERLIN 2012 CONFERENCE

17th-19th October

gbuild:
State of the LibreOffice build system

- Michael Stahl, Red Hat, Inc.

- 2012-10-17

Overview

- The Historic OpenOffice.org Build System
- Goals for a Better Build System
- gbuild Architecture
- gbuild Status
- Lessons Learned
- gbuild Future Work

The Historic OpenOffice.org Build System (1)

- combination of `build.pl` / `deliver.pl` / `dmake`
- `dmake`:
  - conceptually similar to standard make but different syntax
  - OOo the only project using it
  - according to folklore `dmake` was selected in 90s because it was the only thing that worked on Mac OS
  - it's so obsolete it's licensed GPLv1 (!)
- `build.pl` / `deliver.pl`
  - homegrown Perl scripts...

The Historic OpenOffice.org Build System (2)

- `build.pl` iterates over all modules (top-level directories) & invokes `dmake` in each directory
  - obscure `build.lst` files
  - recursive make
    - (technically (almost) no recursion but morally equivalent)
    - “Recursive Make Considered Harmful”, Peter Miller, 1997
    - re -stat lots of files on every `dmake` invocation...
- all `dmakes` in module done: `build.pl` invokes `deliver.pl`
  - copies files listed in `d.lst` to "solver"
    - doesn't "solve" anything (Solar Version)
    - dumping ground for inter-module build

Example:
OOo build (from scratch + running all tests)

- `./configure --enable-foo`
- `./bootstrap`
- `source LinuxX86-64Env.Set.sh`
- `cd smoketestoo_native`
- `Xephyr :42 &`
- `DISPLAY=:42 build --all -P2 -- -P2`
- `DISPLAY=:42 subsequenttests`

Example: OOo build (incremental)

- Let's do some change in vcl...

- `touch vcl/inc/vcl/window.hxx`
- `cd instsetoo_native`
- `build --prepare --from vcl`
- `build --all -P2 -- -P2`

Example: OOo build: clean a single module

- `cd module`
- `deliver -delete`
- `rm -rf $INPATH`

- (alternatively:)

- `cd module`
- `build --prepare --from module`

Example:
OOo build: run subsequenttests in a module

- `cd module`
- `DISPLAY=:42 OOO_SUBSEQUENT_TESTS=t build -P2`

Goals for a Better Build System

- lean prerequisites
  - use standard tools
  - don't want to maintain another `dmake`
- full dependencies for incremental builds
- easy to use & reliable even for non-experts
- easier parallelism, less bottlenecks, better scalability
- less boilerplate in makefiles
- less "creativity" in makefiles
  - there should be one obvious way to to things
- automatically run tests during build
- … all of that with an incremental migration path

Goals for a Better Build System: LO perspective

- LO different from OOo and other OOo based projects:
  - Not large-corporation oriented, but community-oriented
  - *"Every time an incremental build fails a potential contributor is turned away from the project."*
- developers push directly to master, not to feature branches
  - low-level headers tend to change a lot
- incremental builds really have to “just work”!

gbuild Architecture

- one GNU `make` process to build everything
  - but can also build single module
- based on GNU `make` 3.81+ features:
  - `eval`
  - target local variables
- one makefile per deliverable
- full dependencies
  - can be turned off (tinderbox, distro builds)

gbuild Files

- `solenv/gbuild`: core implementation
  - `solenv/gbuild/platforms`: platform specific bits
- `Repository.mk`: define all linktargets/jars
- `RepositoryExternal.mk`: bundled external libs
- `RepositoryFixes.mk`: ugly hacks
- `RepositoryModule.mk`: for single process build
- `config_*.mk`: configure output
- `*/*.mk`: user makefiles

gbuild Implementation

- pseudo-OOP in GNU `make`

  ```
  $(eval $(call gb_Class_method,instance,param...))
  ```

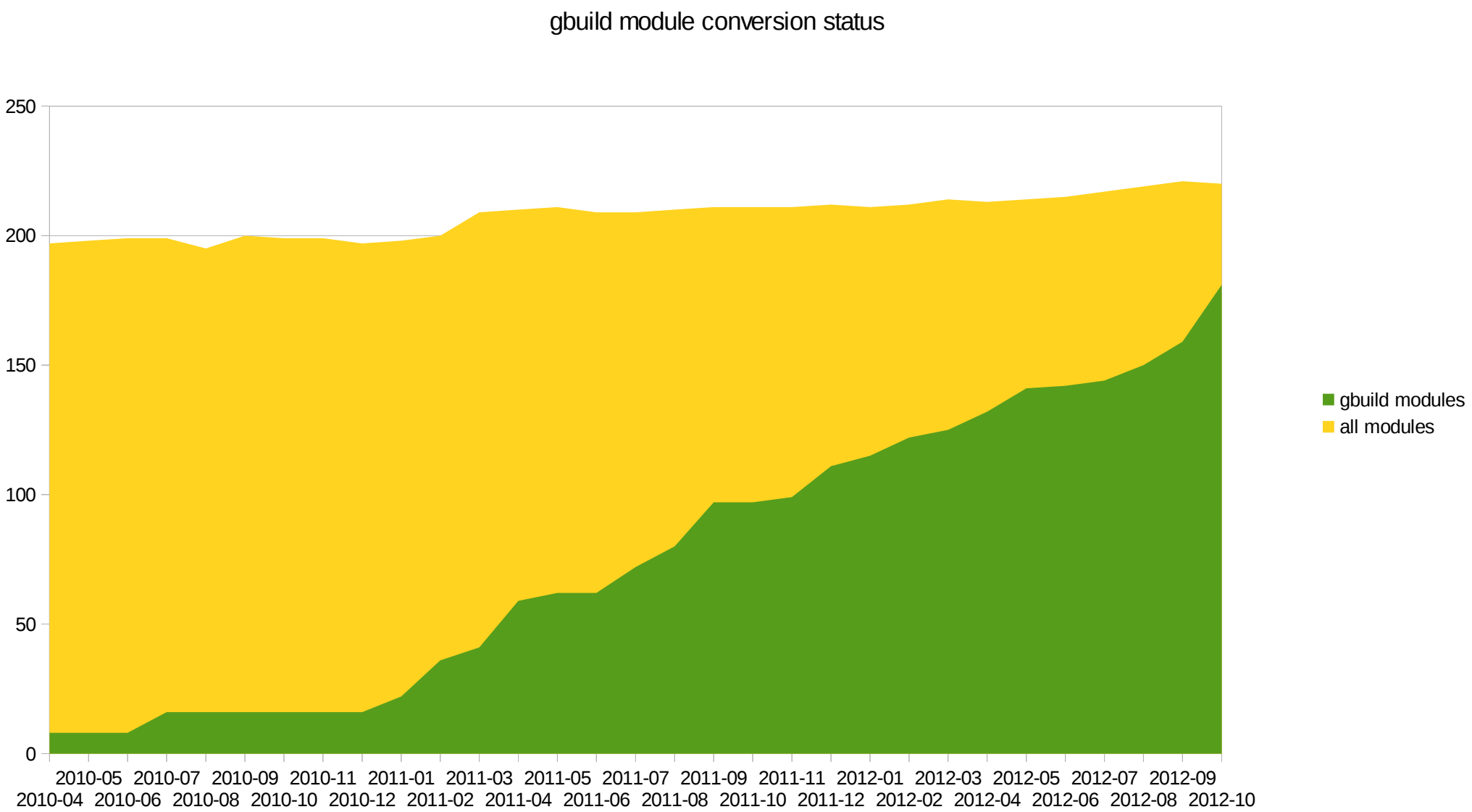
- `solenv/gbuild`: 10k lines of .mk + 200 lines of .awk
- `solenv/gbuild/platform`: 4.5k lines .mk + 200 .awk

- for comparison: `solenv/inc`: 25k of dmake

gbuild Status

- Continuous improvement of core features
- Incremental migration of legacy `dmake` modules

Incremental migration: progress over time:



gbuild New Features (added during last year or so)

- supports standard environment variables like `CPPFLAGS`, `CXXFLAGS`, `LDFLAGS`
- cross compilation support
- new platforms:
  - *BSD, Android, iOS, Solaris/GCC, MSVC2012, AIX
- mergedlibs
- check object owner
- selective debug: `--enable-debug="sw svx xmloff"`
- full dependencies for `svidl`, UNO IDL
- new targets: Asm, Yacc/Lex, Configuration, PyUno, Extension, Dictionary, Scp/InstallModule, Cli, ExternalProject, UI

gbuild is a Community Effort

- thanks to regular contributors:
  - David Tardon
  - Norbert Thiebaud
  - Matúš Kukan
  - Peter Foley
  - David Ostrovsky
  - Bjoern
- and many more than would fit on this slide

Example:
current LO build (from scratch + running all tests)

- `./autogen.sh --enable-foo`
- `make check`

Example: LO build (incremental)

- Let's do some change in vcl...

- `touch vcl/inc/vcl/window.hxx`
- `make`

Example: LO build: clean a module

- `make module.clean`

Example:
LO build: run subsequenttests in a module

- `make module.subsequentcheck`

Example:
LO build: run subsequenttests in a module

- `make module.subsequentcheck`

- … and if it crashes you get a stack trace … automagically!
  - (except if you're unlucky and have to build on Windows... patches welcome)

Bonus Examples: LO build: debugging features

- Run tests in gdb:
  - `GDBCPPUNITTRACE="gdb --args" make`
- Run tests under Valgrind:
  - `VALGRIND=memcheck make module.check`
  - `VALGRIND=memcheck make module.subsequentcheck`
- Run `soffice` in gdb:
  - `make debugrun`

Lessons Learned: Namespace Pitfalls

- everything one `make` process => namespace problems!
  - variable names
    - target local variables not a problem
      - except if initialization forgotten :)
    - prefixes everywhere to avoid collisions
      - gbuild core variables prefixed with gb_
      - variables in user makefiles discouraged
      - user make file variables prefixed with module_
  - pattern rules
    - GNU make 3.81 vs. 3.82 pattern rules
      - some effort to support both

Lessons Learned: Performance

- unwanted parallelism:
  - do not want to link sw in parallel with sd, sc... on your laptop
  - workaround with artificial build order only deps
- portable shell good for performance:
  - dash is faster than bash

Lessons Learned: That Other OS

- Windows makes build system developers unhappy:
  - `make` bug 20033: `make 3.81 -jN` crashy
  - command line length limit
  - cygpath pain
    - finally required `make` with support for DOS paths
  - filesystem, process creation slow...

Lessons Learned: The Good

- full dependencies work!
  - quite simple to extend `svidl`, `idlc` to write make dependencies
- fast no-op builds
- most user makefiles relatively simple
- consistently use `DLLPUBLIC` annotations
- cleaned up cruft like `setsolar`, `set_soenv`... no more shell environment
- sane & consistent way to use external libraries which may be from system or bundled

Lessons Learned: The Not So Good (1)

- core gbuild implementation quite complex and difficult to understand
  - lots of function abstractions
  - `make` is not a very good programming language
  - *"migrating from obscure dmake system to a pile of impenetrable spaghetti masquerading as make files"*
- response files necessary to work around command line length limits on Windows:
  - fortunately `make` 3.83 has grown `$(file ...)` function
- cannot use cygwin's `make` package

Lessons Learned: The Not So Good (2)

- no checking of parameters when calling a function (or that function even exists)
- no multi-target build rules
  - used to work in `dmake`
  - GNU `make` rule can have multiple targets but is invoked once per target then :(
  - requires ugly `touch` rules
- inheritance of target local variables
- evaluating target local variable in dependencies
- bottleneck in parsing? parallelizable?

gbuild Future Work: solver

- solver: an anachronism
  - initially designed for partial builds: only check out a single module from CVS, build that against headers & libraries on NFS share
  - partial builds are obsolete with today's disk sizes
- copying headers around: obsolete, very slow on Windows
  - also breaks incremental builds
- why don't we instead have a runnable LO installation?
  - would obsolete "make dev-install" too...

gbuild Future Work: scp2

- scp2: defines contents of installation sets
  - duplicating a lot of conditionals that are already in makefiles
  - own way to define library names
  - do we still need this? can make do the job directly?

- How many cpps do we really need?
- get rid of more Perl cruft in build system

BERLIN 2012 CONFERENCE

17th-19th October

Thank you for listening

- Questions?