The pkgsrc wrapper framework

Jörg Sonnenberger <joerg@NetBSD.org>

September 19, 2009

Abstract

The wrapper framework in pkgsrc serves two central roles:

- abstracting compiler specifics and
- limiting visibility of installed packages in combination with buildlink.

It helps making package builds a lot more reproducible and decreases the number of patches for platforms that are not using GCC or ELF. The offered flexibility comes at a price, both in terms of execution speed and code complexity. This paper explains how the wrapper framework interacts with the rest of pkgsrc, analyzes the performance of the existing implementation and introduces a simpler and faster reimplementation.

1 Introduction

The NetBSD Packages Collection (pkgsrc) is a cross-platform framework for building and managing software. It is not limited to NetBSD and can be used on a variety of architectures and Operating Systems.

This portability requires supporting different incompatible compiler frontends other than the GNU project C and C++ compiler, GCC. It also requires dealing with the quirks of different shared library formats, especially how the shared library search path is built.

The problem is partially addressed by the wrapper framework. The first important task of the wrappers is to present a GCC-like command line on all platforms. This is done by translating GCC specific options into corresponding options of the target compiler as well as conditionally dropping options as needed.

The second important task of the wrappers is to make builds more reproducible by limiting the visibility of headers and libraries. Most third party programs depend on autoconf or similar mechanisms to identify libraries present and enable features based on the results. E.g. the mplayer package will use a variety of libraries for audio and video codecs if it can find them. This autodetection is often undesirable as it creates hidden dependencies. The buildlink framework is used to express the desired visibility and the wrappers are instructed to enforce this.

For that reason, the wrapper framework is a core component and as it is involved in every single invocation of cc and friends, it has to be fast. The original implementation uses a combination of shell scripts and sed fragments. This requires multiple forks, quite a bit of parsing and adds a noticeable overhead. It was therefore desirable to replace the existing wrapper framework with a faster

code base. A secondary goal was to use an easier to understand structure and to better document what happens in the various phases.

This paper explains the environment created by the buildlink framework, the necessary transformations to make the system compiler cope with this environment and the special problems libtool creates. Last but not least the structure of the new wrappers are discussed and the performance of both implementations is compared. The discussed transformations generally follow the new wrapper, but relevant differences are mentioned.

2 Buildlink

The buildlink framework is one of the core components of pkgsrc. It serves two important purposes. First of all it moves the checking of required versions for dependencies to one single place. As pkgsrc doesn't have the man power e.g. of Debian, it has to make updates a lot less painful for developers. It also means that the amount of testing with mixed versions is more limited and therefore it is desirable to have the common base line in one single place.

The second role of buildlink is to describe the dependencies among packages and more important the required visibility between them. E.g. a package that wants to use GTK+ also needs Glib. Often dependencies are not transitive as programs or libraries are only needed for building or are wrapped and not exposed. Including the buildlink3.mk files of the corresponding package expresses that the content of that package should be visible.

To implement the visibility, a shadow file tree is created in the .buildlink subdirectory of the work area for the package. E.g. for GTK+ the headers in include/gtk-2.0/gtk are symlinked from /usr/pkg to the .buildlink directory. The specific list of the files depends on the content of the package and a list of filters. By default, header files, libraries and pkg-config files are shadowed.

The task of the wrapper framework is now to make sure the compiler is only using the .buildlink shadow tree instead of /usr/pkg by transforming the arguments.

3 Transformations for C-like compiler driver

The transformations are done in multiple phases:

1. Normalization
2. First cleanup
3. Generic option transformation
4. Reordering
5. Compiler-specific option transformation
6. Second cleanup

The phases are run in order and process the result of the previous phase. Each phase can add, change, or drop options.

3.1 The normalization phase

The normalization phase transforms a number of equivalent option forms into a canonical version for each. For example, "-D", "-I" and "-L" can either take the value as separate option or as part of the same option. The normalization also helps some compilers as they don't support the split form.

Linker options ("-Wl,") can be concanated by separating them with ",". To simplify the following processing steps, this is undone first and each component gets a separate "-Wl," prefix.

Library paths can be specified as compiler option with "-L" or as linker option with "-Wl,-L". Rpath entries can be specified by "-R", "-Wl,-R", "-Wl,-rpath" or "-Wl,–rpath". These cases are normalized to "-L" and "-Wl,-rpath" respectively. This normalization makes the following transformation phases simpler as they only have to deal with one canonical form. The old wrappers does not do this transformations and has to deal with the redundancy in different places. The "-Wl,-rpath" and "-Wl,-rpath-link" options are also converted to the single argument form.

Rpath entries can be specified either individually or as PATH style list separated by ":". Such lists are split into individual rpath options to simplify the processing.

Options referring to shared libraries with absolute path names are transformed into a combination of "-L" and "-l". This is necessary on platforms that would otherwise hard-code the shared library path in a way that supersedes the rpath equivalent. It also allows the transformation of both "-L" and "-l" options to apply.

3.2 The cleanup phases

The cleanup phases drop redundant arguments. This reduces the runtime for following phases and helps platforms with small argument size limits.

In this phase, duplicate "-I", "-L", "-Wl,-rpath", and "-Wl,-rpath-link" entries are dropped. The old wrappers dropped duplicate "-D" options too. It doesn't consider "-U" options and macros are generally not duplicated much, so this is of dubious value and was not kept.

Additionally all "-Wl,-rpath" options with relative paths are dropped. Those are sign of buggy build rules and some linkers don't drop them correctly, resulting in surprising behavior or security issues.

Identical consecutive library options ("-l") are unified as well.

3.3 The generic option transformations

This phase first matches the arguments against package specific transformation rules. This is used to optionally drop specific optimizer or warning flags. The old wrappers are performing wildcard matches here and it is not clear yet, if continuing to support that compared to much cheaper fixed expression matches is desirable.

After that "-I", "-Wl,-rpath", and "-L" options with absolute path names are matched against a list of source/destination rules. Source and destination specify the path names. If the path for the option matches one of the rules the source part is replaced with the destination. If no match is found or destination

is empty, the option is dropped. For "-I" and "-L" this is used to redirect the search paths from the installation prefix to the .buildlink shadow copy. For "-Wl,-rpath" it prevents references to arbitrary directories.

Library options ("-l") are searched for transformation rules as well. This allows to replace one library with one or more different libraries. A good example for this is linking against Berkeley DB. Depending on the chosen version, it is called either -ldb, -ldb1, -ldb2, etc. The transformation rules allow the build rules to just use -ldb.

3.4 The reordering phase

The reordering phase exists for the benefit of platforms with stricter library ordering rules than dynamic ELF linkage. This applies especially to static linkage on most platforms and Mach-O linkage on Mac OS X.

The transformation rules in this phase specify that "-lfoo" needs to be after "-lbar". If a mismatch is detected, all instances of "-lfoo" are moved after the first instance of "-lbar", dropping redundant copies.

The old wrappers provide some glue to deal with "-Wl,-Bdynamic" and "-Wl,-Bstatic". This is needed as the old wrappers collect all "-l" arguments after the "-L" arguments. This has created a number of issues over time, the most prominent example of which is the Perl linkage. Thus, this step is not implemented for the new wrappers.

3.5 The compiler-specific option transformations

The compiler-specific option transformation transforms a GCC-ish command line into whatever the platform compiler can understand. All options are matched against a fixed list of rules. The rules are fixed strings or prefix matches. Transformations are typically either pass, drop, or translate.

An example of a more advanced wrapper is the library handling on AIX. The support for "-Wl,-rpath" on AIX exists only on pretty recent versions and is still limited. The wrappers collect all "-Wl,-rpath" options for that purpose and append it as a single "-blibpath" argument with the aggregated result.

4 Dealing with libtool

Libtool is a compiler wrapper to simplify building and using libraries by abstracting platform specific issue. For a cross-platform framework like pkgsrc it is necessary to keep the amount of platform-specific patches manageable. This abstraction comes at a price and one part of that price is the interaction of libtool with the wrapper framework.

Libtool creates three specific issues for linking and installing libraries:

1. Library dependencies can be specified by path names
2. Relinking needs access to newly installed libraries outside the .buildlink shadow tree.
3. Libtool archives (*.la) should not leak references to the .buildlink shadow tree.

To compensate for this a libtool wrapper exists similar to the compiler wrappers.

The relink step enables additional logic in libtool. Whenever it has to relink the output for installation, it keeps two versions of the libtool archive. The first one is the normal ".la" file and includes the relink command. The second version doesn't and is stored with the extension ".lai" in the .libs subdirectory. This is the install version of the libtool archive.

The first problem is addressed by translating direct references to libtool archives similarly to the handling of the "-L" and "-I" options. The second and third problem is addressed by patching the libtool archive and install version after the real libtool command finished.

To allow the relinking step to see the libraries installed by the current package, the library search path is modified for all libtool archives referenced by relative path or an absolute path under the work directory. For each such libtool archive, the .libs subdirectory which contains the real shared and static libraries is added to the search path as "-L" option. This in turn requires that all libtool archives are referenced by path name and not via "-l". As part of the generic transformation phase the libtool wrapper therefore checks all "-l" arguments to see if a corresponding libtool archive exists and is within the work directory. The reference to such a libtool archive replaces the "-l" option.

To prevent accidental leaks of .buildlink references, the options with transformations are processed and the transformations explicitly undone for install version of the libtool archive. This differs from the plain version as by not including the relink command and by not being used to link other local libraries.

5 Reimplementing the wrappers in C

The new wrappers are built on a number of assumptions. These include error handling, reusability, and structure: The wrappers are expected to be reusable. They should have to be built only once or at most once for each compiler if the platform supports more than one. This means that, contrary to the old wrapper code, the wrappers are not fully self-contained. The location of the configuration file is passed down via the environment, which needs care for some packages that alter the environment in unexpected ways.

Error handling is generally straight forward – bailing out. It doesn't make sense to try to recover from memory errors as the compiler will likely fail anyway due to the much higher foot print. Parsing of input files is strict as well, as the configuration file is created by the pkgsrc infrastructure and libtool archives by libtool itself. Strictness is especially important for the latter as only a limited subset of shell logic is implemented. Another import area for sanity checking is the command line. GCC itself is often quite forgiving for obvious mistakes. For example "-L" and "-I" options without arguments are silently dropped. This can result in interesting bugs when reordering is done. Forgetting the argument for "-Wl,-rpath" can result in obscure linker errors. Being more aggressive here helps to identify errors.

The wrappers share a common work flow:

1. Parse configuration file
2. Convert command line arguments into a list

3. Log original command
4. Run the transformation phases in order
5. Log modified command
6. Fork if needed and execute the modified command
7. Do post-processing for the libtool wrapper

The transformation phases separately iterate over the arguments, dropping, adding, or replacing arguments as they go. The cleanup phases are utilizing hash tables to find duplicate options. The compiler-specific transformations are using a perfect hash table for all fixed options and iterate the remaining few rules in order. The implementation considering options as mostly inmutable strings. This increases the pressure on the malloc implementation to use pools for common allocation sizes.

The fork in the wrapper is only done if there is processing after the exit. Depending on the architecture, this is one of the slowest operations possible, so avoiding it can help a lot.

Overall, the current new wrapper consists of approximately 64KB or 2600 lines of code.

To evaluate the performance, the pkgsrc framework has been modified to conditionally support both old and new wrappers. The only difference for execution is the directory passed in the PATH variable.

The first test case is pkg_install. This is excersising the normal compiler wrappers, but not libtool. The source files to compile are relatively small. The overhead for the new wrapper is one exec per compiler invocation and the processing itself. No fork is required for this case.

The second test case is libX11. This adds libtool processing for all three cases. The elimination of redundant options explains why using the new wrappers is faster than not using wrappers at all. As the libtool wrapper has to post-process the output files, it has to fork.

The third test case is cmake. This differs from the pkg_install test case in the complexity of the compiled files. As the source consist of larger C++ files, the overall runtime is dominated by the C++ compiler.

6 Conclusion

The new wrapper framework has proven a success by identifying issues that where not visible before as well as improving the over-all pkgsrc experience. The performance advantage is very noticeable on platforms with cheap fork and can be expected to be even bigger on slower embedded systems.

Further work will focus on integrating the new wrappers into the main tree. This includes porting the platform and compiler specific code from the old wrapper framework. Another area of interest is profiling based on the usage in larger packages to evaluate the fitness of the chosen algorithms.

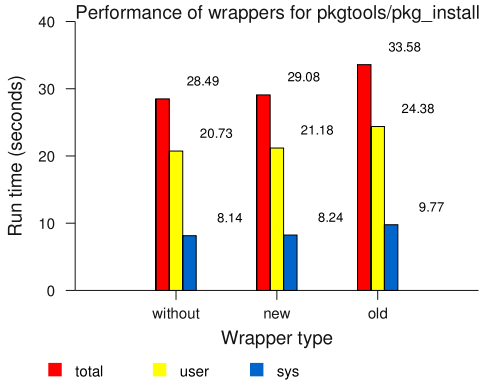

Figure 1: Comparing wrapper performance for pkgtools/pkg_install

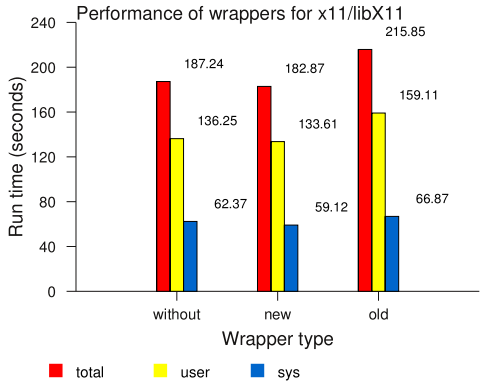

Figure 2: Comparing wrapper performance for x11/libX11

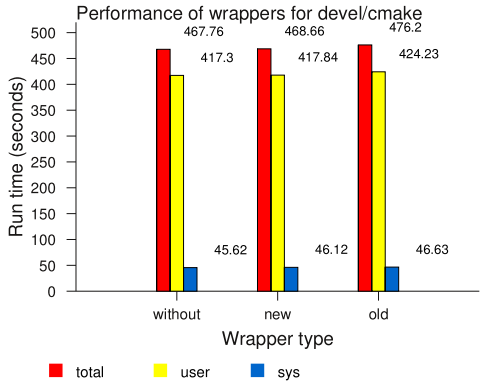

Figure 3: Comparing wrapper performance for devel/cmake