Multi-Arch
Implementation Strategy

Prepared for: HP

By: Canonical Limited

May 2, 2006

i n v e n t

1 Introduction

Many modern processors are able to run binaries intended for architectures other than their own native architecture, for example new Intel processors are able to run binaries built for the `i386`, `amd64` (aka. `x86_64`), `ia64` and `hppa` architectures.

Ordinarily this simply gives a freedom of choice as to which operating system architecture to install on the system, however with the proper kernel and library support it is possible for software from all possible architectures to be freely intermixed on the same system. As such systems are able to run binaries from multiple architectures side-by-side, we use the term *Multi-Arch* to describe them.

2 Brief

This report investigates the various possible ways to extend Debian and derived-distributions to provide simple Multi-Arch support to system administrators who desire it. The desired final implementation can be trialled in Ubuntu during the development cycle of an upcoming release and is intended to be deployed in the Debian *etch* release.

Also a study into the feasibility of Multi-Arch is performed by creating the desired environment by hand from the list of packages currently in the `ia32-libs` package, given in Appendix A. In particular this attempts to determine how much of the necessary file relocation can be performed automatically and how much manual work will be required for each source package.

Our conclusions and recommendations are given in Section 7.

3 Problems

This section outlines the general problems with creating a Multi-Arch system that need to be overcome by proposed implementations.

3.1 Architecture-dependent library locations

The primary problem to be overcome is that of the location of architecture-dependent files, in particular the shared libraries normally located in the `/usr/lib` directory. As shared libraries are dependencies of many packages, it is highly probable that the same shared library for two different architectures would need to be installed at the same time. It is not possible for an `amd64` application to

load an i386 library due to differences in such things as address space, calling conventions, word and data sizes, etc.

The i386 and amd64 packages of libx11-6 for example both ship the /usr/lib/libX11.so.6.2.0 library and its /usr/lib/libX11.so.6 symlink. If both were attempted to be installed at the same time, these files would conflict.

A proposed, and highly-favoured, solution for this is to move libraries into sub-directories of /usr/lib named after the architecture they are intended for. The i386 linker and dynamic link-loader would look in /usr/lib/i486-linux-gnu and the amd64 linker and link-loader would look in /usr/lib/x86_64-linux-gnu for their libraries.

Thus the library locations would not conflict, and depending binaries would be able to load the library for their own architecture without getting the wrong one by mistake.

3.2 Architecture-dependent binary locations

An additional problem is that of the architecture-dependent binaries themselves, as these are located in /usr/bin and /usr/sbin for all architectures.

However unlike shared libraries, which are architecture-specific dependencies of many different packages, binaries are usually the final desired product and not a dependency of another. Binaries sometimes can be a dependency of another package, but unlike shared libraries generally the conventions are that of the binary and are architecture-independent.

It's generally possible, and indeed in some cases desirable, for binary or library of one architecture to use a binary from another. An i386 package could use the openssl binary from the ia64 architecture which is better optimised for the system.

For this reason the general solution is to simply not allow the same binary from two different architectures to be installed at the same time, and treat attempts do so as either conflict or intent to change the architecture of the installed package.

System administrators wishing to do so for the own reasons would unpack the package by hand into a different location such as /opt or /usr/local.

3.3 Architecture-independent files

Architecture-independent files are usually located in directories under /usr/share. These generally consist of support files for the libraries as well as all-important documentation.

Fortunately these files should be identical for all architectures, so there is no need to relocate them or install multiple copies; all of the installed architectures can share the same files.

Implementations will still need to specify a way to avoid the same file existing in both packages from conflicting, which would be dpkg's ordinary operation in such circumstances.

Often a suggestion for this is for policy to forbid architecture-independent files from being present in architecture-dependent library packages; not in the least because of the waste of archive space that generally results. This isn't an absolute solution though as the files in question could be sufficiently small that splitting them out would not be a benefit.

3.4 Hard-coded paths

A problem not directly related to file conflicts is that of hard-coded paths to architecture-dependent files, often in the libraries themselves. Often these are things such as locations of plugins to be loaded with `dlopen` or similar.

There's no reliable automatic way to tackle these, so the only solution available is to address it with policy and either forbid such paths or require that the packages themselves be manually made aware of the Multi-Arch locations they will be installed to.

Where hard-coded paths exist in architecture-independent files, that would also have to be tackled with policy and such files be declared illegal. A typical example of this is `libpango1.0-0` which places the paths to module libraries in `/etc/pango/pango.modules`. As these paths are architecture-dependent, this file could not be shared by multiple architectures.

Suggested solutions for that particular case include renaming or moving the file to a path that includes the architecture name or removing the full paths from the file and using a path relative to one known by the library itself.

Other cases would have to be addressed on an individual basis.

3.5 Dependencies

A further problem is dealing with the dependencies of a particular package in the Multi-Arch environment. As is noted above when discussing the difference between a shared library and a support binary, the dependency can be architecture-specific or open.

As the libraries and support binaries could be in other packages on the system the dependency system would need to be flexible enough to allow the package maintainer to express whether the dependency is architecture-specific or not.

Where a dependency is architecture-specific the package manager would have to ensure that a package of the same architecture is installed on the system; if that package isn't Multi-Arch capable, it would need to effectively conflict with any installed package of a different architecture.

For open dependencies on shared content, the package maintainer would have to be more careful to ensure that the correct version of the shared files are installed for any possible architecture of the binary or library package. This is already largely taken care of with the versioned dependencies they already declare to prevent the common package from being too old for the binary, or vice-versa.

A caveat occurs when the common package declares a versioned dependency on a binary package, the package manager would have to ensure the binaries of all installed architectures meet the requirements; not just one of them.

4 Requirements

While the outcome of the different possible implementations are all different to various extents, there are a set of requirements that the winning implementation should show. These include:

- Special packages should not be required, it should be possible for a system administrator to simply download a library package from another supported architecture and install it on their system.
- Minimal use of disk space is also desired, where possible files that can be shared between the different architectures should be shared.
- Shared configuration files should be used, it should not be necessary to modify configuration files for each architecture but one global set for all of the installed software.
- Impact on package maintainers to be as small as possible, as any implementation that requires extensive modifications to source packages will not likely gain favour with those who have to make the changes.
- Easy for the system administrator to take advantage of, without requiring any special knowledge of how the system works.

5 Possible Implementations

The long term solution is for upstreams and thus the package mantainers to alter their packages to install libraries into the Multi-Arch compatible paths, such as `/usr/lib/i486-linux`.

This section describes possible short-term solutions for accelerating the usefulness of these systems, and also working around packages not yet converted in the medium term.

5.1 Chroots

By far the easiest way to install software from multiple architectures on a single system is to use chroots, each with the essential packages and dependencies of binaries for that architecture installed into them. Home directories, etc. can be shared by using bind mounts, however it is not advisable to attempt to do this for directories such as `/etc` or `/usr/share` due to the damage the package manager in the chroot could do to the real system.

Binaries could be "integrated" into the main system using a shell wrapper that alters the kernel personality, relocates into the chroot, then runs the intended binary with the expected arguments. Such a wrapper could be as simple as:

```
#!/bin/sh

exec linux32 dchroot -c i386 "$0" ${1+"$@"}
```

While this implementation is certainly the easiest, it doesn't meet many of the requirements due to the large waste of disk space and the increased administration cost of creating and maintaining the wrappers by hand.

Advantages:

- No development work required.

Disadvantages:

- Massive over-use of disk space.
- Requires per-architecture configuration.
- Lots of work for administrator.

5.2 Environment packages

The current approach to Multi-Arch is to generate very large packages such as `ia32-libs` which themselves contain the sources to various essential and useful library packages compiled for the different architecture and installed into `/usr/lib64`.

While these are easy for a systems administrator to deploy on their system, they only provide the libraries and no support for installing binaries for that architecture is catered for. A system administrator wanting to install the `i386` version of `mplayer` on their system would have to do so by hand; including any dependent libraries not provided by `ia32-libs`.

Other problems with this implementation include the rate at which the sources in the package go out of date, leading to both outdated binaries and a nightmare in dealing with security updates and the sheer size of the package.

Advantages:

- Minimal package maintainer impact.
- Easier for system administrator than chroots.

Disadvantages:

- Packages are hard to maintain.
- Security updates are a nightmare.
- No provision for binaries from other architectures.
- Inflexible

5.3 Multiple binary production

An increasingly popular trend is to make source packages produce binary packages for multiple architectures which can support it. The `zlib` source package, for example, produces `zlib1g` (native) and `lib32z1` binaries on `amd64` and `zlib1g` (native) and `lib64z1` binaries on `i386`.

This approach solves some of the biggest problems with the `ia32-libs` style approach detailed in Section 5.2 at the cost of putting the burden onto the package maintainers.

The same can be done for binary packages that are intended to be available for different architectures, much as it is for the kernel. This approach can either

adopt the ia32-libs style /usr/lib64 paths or the ideal Multi-Arch style /usr/lib/x86_64-linux paths.

Advantages:

- Package maintainer is aware of Multi-Arch concerns.
- Separate compilation means hard-coded paths are likely to be changed.

Disadvantages:

- More than doubles archive space for every Multi-Arch package, lib32z1_*_amd64.deb and zlib1g_*_i386.deb are theoretically the same.
- Burden of large amounts of work is placed on package maintainers.
- Dependencies become more difficult to keep track of as the numbers of library packages doubles.

5.4 Automated package rewriting

Instead of relying on the package maintainers to modify their packages to produce multiple binaries, it is possible to automatically produce them from the binaries intended for the other architecture. For example, a zlib1g-i386_*_amd64.deb package could be produced from the zlib1g_*_i386.deb package.

Architecture-dependent files in the package would be moved to the relocated path and architecture-independent files simply removed from the alien package. Packages with hard-coded paths would not be able to be modified, and thus would fail to work and require source changes.

As the package names would change, control files in the package would also need to be rewritten to adjust dependencies to the new names. There would therefore need to be rules which defined which packages would be rewritten and which wouldn't, for example all packages matching lib* could be changed to lib*-ARCH.

Advantages:

- Requires no modification to dpkg as packages given to it are for the native architecture.
- Works for packages containing binaries as well as libraries.

Disadvantages:

- Packages with hard-coded paths fail to work.
- Rewriting dependency fields is not flexible, there is no way to specify whether a dependency is architecture-specific or not.
- No fool-proof way to avoid renaming architecture-independent common packages that are often also named `lib*`, both for the packages themselves and their appearances in dependency fields.
- Provides, Conflicts and Replaces are a problem and need special handling.
- `glibc` needs to be special-cased.

This rewriting can be done at two different points, each with its own additional advantages and disadvantages.

5.4.1 At build time

The rewriting script would be part of the buildd software and would automatically produce Multi-Arch packages for all libraries and binaries in its white-list.

Advantages:

- Most of the work is performed behind the scenes.
- Requires no modifications to `APT`.

Disadvantages:

- More than doubles space required on the archive.

5.4.2 At install time

The rewriting script would be run by APT after downloading any packages from an alien architecture to convert the package into one that it can give to `dpkg` to be installed.

Advantages:

- Archive only contains one copy of each package for each architecture.

Disadvantages:

- Requires modifications to `APT` to be able to process two architectures and their dependency trees, and merge it with its own.
- Hard to blacklist packages that can't be made Multi-Arch compatible.

5.5 Modify dpkg 1.13

Many of the problems with Multi-Arch can be solved by extending the package manager itself, dpkg, to support a system with packages from multiple architectures installed on it.

The dependency engine would need to be extended to support multiple trees that overlap and share common architecture-independent packages. At the same time it should probably be extended to allow the package maintainer to specify in the control file whether a dependency is for a package of the same architecture, or for any architecture.

The package database, and associated logic, would need to be changed to allow the same version of a package to be installed for multiple architectures, while at the same time keeping track of the potentially differing file lists and dependencies for the two packages.

It would need to be decided how to handle the possibility of different versions of a given package from different architectures being on the system at any one time. The suggested solution is to require that all of the installed architectures of a package are at the same version, in effect, a package declares an absolute versioned dependency on itself. dpkg's upgrade mechanism would handle this cleanly by not configuring the packages until they are all unpacked, and doing them all at once.

The unpacking routines would need to be modified for when they encounter a file that exists in both packages, which by policy would only be permitted to be identical shared files. Paths to libraries themselves would need to be changed in the source package to the Multi-Arch style paths.

Advantages:

- Native packages from desired architectures can be used.
- Control file syntax can be extended to express whether a dependency is architecture-specific or not.
- Hard-coded paths solved by putting that responsibility onto the package maintainer to fix as they relocate paths.

Disadvantages:

- Potentially large amounts of work to a cryptic and poorly understood code-base.
- Also requires modifications to APT.

- Handling for virtual packages, and Provides/Conflicts/Replaces would need to be decided.
- Source packages need to be modified to put libraries into new paths.

5.6 Implement dpkg 2.0

The proposed design for `dpkg` 2.0 offers many attractive features that would be useful for implementing a Multi-Arch system that could truly use the same binary packages without modification.

5.6.1 Architectures

Architectures are expressed as a series of dependencies, rather than a string that is directly compared for equality. An ordinary home system might provide just "`i386`" while a Multi-Arch capable server would provide "`amd64 i386 ia64 hppa`". Packages would simply depend on the appropriate string for their architecture, so an `i386` package can be installed on either system.

The architecture tag in the package filename would be for reference only, and to aid in FTP archive location. It is chosen from a table mapping tags to dependency lists, out of all the tags whose dependencies are all shared by the package the most specific one is chosen to represent the package.

A proposal is to extend this to include additional system features, for example processor flags such as `sse` and other later instruction sets, our example home desktop might actually provide "`i386 i486 i586 i686 sse`". Packages that would benefit from can then be compiled and depend on these features, and located in a *small architecture* with only a tiny handful of other packages. One might then install `mplayer_1.0-1_i686+sse.deb` onto their system, in effect, every system becomes a Multi-Arch one.

5.6.2 File meta-data

Detailed information about each file is stored in the package database, as well as the information about the package itself. This makes it far easier to track files that only exist for a particular architecture, or are shared between multiple packages. Such things cannot trivially be done with the current `dpkg` implementation because it keeps no such data.

5.6.3 Classes

The actual work of installing a file onto the system is performed by things known as *classes*, either code in `dpkg` itself or a script or binary provided in the package or installed by the system administrator.

When a package is to be unpacked or removed, the files are placed in a temporary location and the class script is called with the meta-data about the files on standard input. The job of the script is to move the file into place, and make such adjustments to it that the meta-data suggests.

Because classes may be stacked and chained, much of the default handling is taken care of by code in `dpkg` itself and can also be extended by the administrator (e.g. for SELinux).

The reason these are useful for Multi-Arch is that a class script itself is the policy for how a file is moved into position, and may if it desires chose not to do so and indicate that it is user-modified, shared or has otherwise vanished.

This means that the architecture-independent files in Multi-Arch packages could be marked as belonging to a class script that only installs the file if it does not exist already, and doesn't conflict if it does and is owned by the same package of a different architecture.

5.6.4 Filters

Before a package is unpacked, the meta-data for each file can be altered by *filters*, code in `dpkg` itself or a script or binary provided in the package or installed by the system administrator, just as with classes.

Filters are stacked and chained like classes and are called with the meta-data about the files on standard input, the filter may make changes by sending the new data to standard output. Changes may be made to any information about the file, including its intended destination path and which class is in.

Multi-Arch systems could include a filter that automatically relocated libraries from `/usr/lib` into the Multi-Arch directory to force support for libraries that have not yet had the changes made at the source package level.

The filter could also change the class of the shared files to that which does not produce a conflict if the file already exists.

Advantages:

- All `i386` packages could theoretically be installed on `amd64`, even those which haven't had any work done on them yet (hard-coded paths permitting, of course).

- No duplication in the archive.
- Control file syntax can be extended or changed to suit.
- Potentially easier to write than trying to modify existing `dpkg` code.

Disadvantages:

- Potentially expensive to develop and test.
- Requires modifications to `APT` and potentially all tools that go near `dpkg`.

6 Feasibility Study

The purpose of this feasibility study is to determine whether the majority of packages are able to be automatically converted into Multi-Arch capable packages.

To test this we attempted to get OpenOffice.org 2 running on a Multi-Arch system, this was a logical choice given that it is the primary package for which the current `ia32-libs` package family exists.

6.1 Process

The dependency packages required for this study are listed in Appendix A. The script we used to perform the automatic relocation and control file rewriting is given in Appendix B.

The script was run on all of the `i386` library packages, it renames the package appending "`-i386`" to the name and changing it to the local architecture. Dependencies are rewritten such that all dependencies on library packages (matched using a primitive regular expression) also have "`-i386`" appended to the package name. Files not desired in the package are removed, and those that need to be relocated are moved, then the package is created again for installation.

In a production system this script would need to be run over the native architecture packages too, also an additional script would need to be run that just performed the dependency rewriting for all packages on the system.

6.2 Problems encountered

While the script we used was not intended to be perfect, it yielded a fair number of corner cases which would not only affect any real attempt to rewrite packages but also affect automatic rewriting by a package manager.

6.2.1 Architecture-independent package names

One major problem was distinguishing architecture-independent packages from architecture-dependent ones from the name alone; for example the `libgtk2.0-common` package.

This package is architecture-independent yet the test script renamed it to `libgtk2.0-common-i386` and made it into an `amd64` package which contained no files.

Whilst solving this when rewriting the package itself is trivial, as the nature of the package is revealed in both the filename and the "Architecture" field in its `control` file; solving this when encountering the package name in another's dependency field is not so easy.

This problem is unique to package rewriting; if the package manager was changed or rewritten to allow multiple simultaneous installation of the same package name, it would be not be necessary to rename them like this.

6.2.2 Virtual packages

Another problem we encountered was dealing with the Provides, Conflicts and Replaces dependency fields, in particular their use with virtual packages; for example the `libldap2` package provides and conflicts with `libldap-tls`.

When a library package provides a virtual package, and either replaces or conflicts with it, it is also replacing or conflicting with itself from a different architecture.

Attempting to install both `libldap2-amd64` and `libldap2-i386` together resulted in a package conflict, because both conflicted with a virtual package the other provided.

The only solution to this problem would appear to be handling inside the package manager itself; `dpkg` already exempts a package from conflicting with itself, that handling would be extended to include all copies of itself from different architectures currently installed.

6.2.3 Maintainer scripts

An unfortunate number of packages make changes to their library directories in their `postinst` maintainer script, including creating, moving and removing symlinks to libraries.

Another particularly obvious problem is that of calling `ldconfig`, packages would need to call the right version for their architecture to update the right

cache. Alternatively this would have to be replaced by a wrapper that ran `ldconfig` for every architecture for which there was a dynamic link-loader.

6.2.4 Hard-coded paths

As noted earlier, hard-coded paths were a significant problem. A particular problem-package we encountered was `pango` which lists absolute paths to shared libraries in its `/etc/pango/pango.modules` file.

Unfortunately there are no clean solutions to this problem; a notable unclean one would be to use a small wrapper library to divert file access from `/etc/pango/pango.modules` to `/etc/pango/pango32.modules` which would be a copy of the former with the plugin paths changed.

This solution does not scale, except for single use-cases such as making OpenOffice.org run.

The only real solution is to fix the packages themselves to not hard-code paths in such manner.

7 Conclusions

Neither chroots or environment packages such as `ia32-libs` truly tackle the Multi-Arch problem or solve it in a desirable way, so while they are tempting from a pure cost point of view, we do not recommend them as a viable option.

Production of multiple binaries wastes a large amount of archive space and puts all of the responsibility on the package maintainers themselves. This actually has the largest amount of work that needs to be done, so also has the highest cost; for this reason we do not recommend it.

Automated package rewriting seems tempting at first glance, however our trials have uncovered significant problems which can only truly be overcome by modification of the package manager.

Our recommendation therefore is to place Multi-Arch support directly into the package manager, either by modification of the existing `dpkg 1.13` code or by developing `dpkg 2`. While in the short term developing and testing `dpkg 2` may seem more expensive and thus not the best option, we believe that in the long term it will ultimately prove a better course than retro-fitting support into `dpkg 1.13` as all of the problems can be met head-on.

A Study Environment

Below are the lists of source packages contained in the `ia32-libs` package and other related packages in Ubuntu dapper, used to construct a Multi-Arch test environment.

A.1 ia32-libs

Source package	Version in dapper
acl	2.2.34-1
attr	2.4.25-1
bzip2	1.0.3-0ubuntu1
coreutils	5.93-5ubuntu2
db4.3	4.3.29-3ubuntu2
expat	1.95.8-3
fontconfig	2.3.2-1.1ubuntu3
freetype	2.1.10-1ubuntu1
gcc-3.3	3.3.6-10
glibc	2.3.6-0ubuntu8
lcms	1.13-1
libcairo	1.0.2-3ubuntu1
libdrm	2.0-0ubuntu1
libice	1.0.0-0ubuntu2
libidn	0.5.18-1
libjpeg6b	6b-11
libpng	1.2.8rel-5
libsm	1.0.0-0ubuntu2
libsndfile	1.0.12-3
libwmf	0.2.8.3-3.1
libx11	1.0.0-0ubuntu3
libxau	1.0.0-0ubuntu3
libxaw	1.0.1-0ubuntu2
libxdmcp	1.0.0-0ubuntu2
libxext	1.0.0-0ubuntu3
libxi	1.0.0-0ubuntu2
libxinerama	1.0.1-0ubuntu2
libxml2	2.6.23-1ubuntu4
libxmu	1.0.0-0ubuntu3
libxp	6.8.2-11ubuntu2
libxpm	3.5.4.2-0ubuntu2
libxrandr	1.1.0.2-0ubuntu2
libxrender	0.9.0.2-0ubuntu2
libxt	1.0.0-0ubuntu2

libxtrap	1.0.0-0ubuntu2
libxtst	1.0.1-0ubuntu2
libxxf86vm	1.0.0-0ubuntu2
linux-kernel-headers	2.6.11.2-0ubuntu16
mesa	6.4.1-0ubuntu6
ncurses	5.5-1ubuntu3
pam	0.79-3ubuntu10
popt	1.7-5
tiff	3.7.4-1ubuntu1
xaw3d	1.5+E-9ubuntu1
zlib	1.2.3-6ubuntu4

A.2 ia32-libs-gtk

Source package	Version in dapper
atk1.0	1.11.2-0ubuntu1
gconf2	2.13.5-0ubuntu4
glib2.0	2.10.1-0ubuntu1
gtk+2.0	2.8.14-0ubuntu1
gtk2-engines	2.6.7-0ubuntu1
libpixman	0.1.6-1
libxcursor	1.1.5.2-0ubuntu2
libxfixes	3.0.1.2-0ubuntu2
libxft	2.1.8.2-0ubuntu2
orbit2	2.13.3-0ubuntu1
pango1.0	1.11.99-0ubuntu2
ubuntulooks	0.9.5-1

A.3 ia32-libs-openoffice.org

Source package	Version in dapper
curl	7.15.1-1ubuntu1
cyrus-sasl2	2.1.19-1.7ubuntu4
e2fsprogs	1.38-2ubuntu1
firefox	1.5.dfsg+1.5.0.1-1ubuntu7
flac	1.1.2-3ubuntu1
gnutls12	1.2.9-2ubuntu1
icu	3.4-4
krb5	1.4.3-5
libart-lgpl	2.3.17-1
libgcrypt11	1.2.2.-1
libgpg-error	1.1-4

libtasn1-2	0.2.17-1ubuntu1
libwpd	0.8.4-2
libxslt	1.1.15-1ubuntu1
mdbtools	0.5.99.0.6pre1.0.20051109-2.1ubuntu3
nas	1.7-3ubuntu3
neon	0.25.3.dfsg-1
openldap2	2.1.30-12ubuntu3
openssl	0.9.8a-5
portaudio	18.1-4
startup-notification	0.8-1ubuntu1
stlport4.6	4.6.2-3
xmlsec1	1.2.9-1ubuntu1

B Package Rewriting Script

```
#!/bin/sh

error() {
    echo "$@" 2>&1
    exit 1
}

arch="$(dpkg --print-architecture)"

[ $# -gt 0 ] || error "Package filenames must be specified"

tmpdir="$(mktemp -d)"
trap "rm -rf $tmpdir" 0

for package; do
    [ -f "$package" ] || error "$package: no such file"

    # Extract package information from the name
    pkginfo="$(basename $package)"
    pkginfo="${pkginfo%.deb}"
    pkgname="${pkginfo%%_*}"
    pkginfo="${pkginfo#*_}"
    pkgver="${pkginfo%_*}"
    pkgarch="${pkginfo#*_}"

    triplet="$(dpkg-architecture -a${pkgarch} -qDEB_HOST_GNU_TYPE 2>/dev/null)"

    # Unpack the package
```

```
pkgtmp="$tmpdir/${pkgname}_${pkginfo}_${pkgarch}"
mkdir "$pkgtmp"
dpkg-deb -e "$package" "$pkgtmp/DEBIAN"
dpkg-deb -x "$package" "$pkgtmp"


# Rewrite the Package field
perl -p -i -e "m/^Package:/ and s/\b(\Q$pkgname\E)\b/\1-$pkgarch/" \
        "$pkgtmp/DEBIAN/control"

# Rewrite the Architecture field
sed -i -r -e "/^Architecture:/s/\b$pkgarch\b/$arch/" \
        "$pkgtmp/DEBIAN/control"

# Rewrite the dependency fields
for field in Pre-Depends Depends Recommends Suggests Replaces \
             Conflicts Enhances; do
    sed -i -r -e "/^$field:/s/\b(lib[^ ,]*)/\1-$pkgarch/g" \
            "$pkgtmp/DEBIAN/control"
done

# Stick multi-arch comment in the description
sed -i -r -e "/^Description:/s/$/ ($pkgarch multi-arch package)/" \
    "$pkgtmp/DEBIAN/control"


# Remove anything that's not a library
for discard in /usr/X11R6 /usr/bin /usr/doc /usr/games /usr/include \
               /usr/local /usr/man /usr/sbin /usr/share /usr/src \
               /sbin /bin ; do
    rm -rf "$pkgtmp/$discard"

    sed -i -r -e "\| ${discard#/}/|d" "$pkgtmp/DEBIAN/md5sums"
done

# Relocate libraries
for relocate in /lib /lib32 /lib64 /usr/lib /usr/lib32 /usr/lib64; do
    [ -d "$pkgtmp$relocate" ] || continue

    mv "$pkgtmp$relocate" "$pkgtmp/DEST"

    dest="$pkgtmp${relocate%[0-9]*}"
    [ -d "$dest" ] || mkdir -p "$dest"
```

```
        mv "$pkgtmp/DEST" "$dest/$triplet"
    done


    # Build the package again
    dpkg-deb -b "$pkgtmp" "$(dirname $package)/"\
                "/${pkgname}-${pkgarch}_${pkgver}_${arch}.deb"
done
```