Overhauling Amd for the '00s: A Case Study of GNU Autotools *

Erez Zadok
Stony Brook University
ezk@cs.sunysb.edu

Abstract

The GNU automatic software configuration tools, Autoconf, Automake, and Libtool, were designed to help the portability of software to multiple platforms. Such *autotools* also help improve the readability of code and speed up the development cycle of software packages. In this paper we quantify how helpful such autotools are to the open-source software development process. We study several large packages that use these autotools and measure the complexity of their code. We show that total code size is not an accurate measure of code complexity for portability; two better metrics are the distribution of CPP conditionals in that code and the number of new special-purpose Autoconf macros that are written for the package.

We studied one package in detail—Am-utils, the Berkeley Automounter. As maintainers and developers of this package, we tracked its evolution over ten years. This package was ported to dozens of different platforms and in 1997 was converted to use GNU autotools. We show how this conversion (autotooling) resulted in a dramatic reduction in code size by over 33%. In addition, the conversion helped speed code development of the Am-utils package by allowing new features and ports to be integrated easily: for the first year after the conversion to GNU autotools, the Am-utils package grew by over 70% in size, adding many new features, and all without increasing the average code complexity.

1 Introduction

Large software packages, especially open-source (OSS) ones, must be highly portable so as to maximize their use on as many systems as possible. Past techniques for ensuring that software can build cleanly and run identically on many systems include the following:

- Asking users to manually configure a package prior to compilation by editing a header file to turn on or off package features or to specify services available from the platform on which the package will be run. This process required intimate knowledge of the system on which the package (e.g., C-News) was to be built.
- Trying to achieve portability using CPP macros and nested `#ifdef` statements. Such code results in complex, system-specific, deeply-nested CPP macros which are hard to maintain.
- Using Imake [2], a system designed specifically for building X11 applications. Imake defines frozen configurations for various systems. However, such static configurations cannot account for local changes made by administrators.
- Using Metaconfig [11], developed primarily for building Perl. This system executes simple tests similar to Autoconf, but users are often asked to confirm the results of these tests or to set the results to the proper values. Metaconfig requires too much user interaction to select or confirm detected features and it cannot be extended as easily as Autoconf.

GNU Autoconf [5] solves the above problems by providing canned tests that can dynamically detect various features of the system on which the tests are run. By actually testing a feature before using it, Autoconf and its sister tools Automake [6] and Libtool [7] can build packages portably without user intervention. These automated software configuration tools (*autotools* [10]) can run on numerous systems. Autotools work identically regardless of the OS version, any local changes that administrators installed on the system, which system software

* Appears in proceedings of the 2002 Annual Usenix Technical Conference, FreeNIX track.

packages were installed or not, and which system patches were installed.

The rest of this paper is organized as follows. Section 2 explains the motivation for automated software configuration tools. In section 3 we explain how GNU Autoconf and associated tools work, explore their limitations, and describe how we used these tools. Section 4 evaluates several large OSS packages and details the use of autotools in the Am-utils package. We conclude in Section 5.

2 Motivation

When software packages grow large and are required to work on multiple platforms, they become more difficult to maintain without automation. We spent several years maintaining Amd and Am-utils, as well as fixing, porting, and developing other packages. During that time we noticed how difficult it was to maintain and port such packages and that led us to convert Amd to use autotools. As a result of the conversion, we noticed that Am-utils became easier to maintain and port. We therefore set out to quantify this improvement in the portability and maintainabilty of the Am-utils package, and those investigations led to writing this paper.

There are six reasons why porting such packages to new platforms, adding new features, or fixing bugs becomes a difficult task more suitable for automatic configuration:

1. **Operating system variability**: There are more Unix systems available today, with more minor releases, and with more patches. Flexible software packaging allows administrators to install selective parts of the system, increasing variability. An automated build process can track small changes automatically, and can even account for local changes.
2. **Code inclusion and exclusion**: To handle platform-specific features, large portions of code are often surrounded by `#ifdef` directives. Platform-specific code is mixed with more generic code. Often, system-specific source files are compiled on every system, because there is no automatic way to compile them conditionally.
3. **Multi-level nested macros**: To detect certain features reliably, older code uses deeply nested `#ifdef` directives. This results in complex macro expressions designed to determine features as reliably as possible. The main problem with such macros is that they provide second-hand or anecdotal knowledge of the system. For example, to test if a compiler supports "void *", some code depends on the name of the compiler (GNUC) rather than directly testing for that feature's existence.
4. **Shared libraries**: Many packages need to build and use shared or static libraries. Such packages often support shared libraries only on a few systems (e.g., Tcl before it was autotooled), because of differing shared library implementations. Frequent use of non-shared (static) libraries results in duplicated code that wastes disk space and memory.
5. **Human errors**: Manually-configured software is more prone to human errors. For example, the first port of Amd to Solaris on the IA32 platforms copied the static configuration file from the SPARC platform, incorrectly setting the endianness to big-endian instead of little-endian.
6. **Novice and overworked administrators**: With a rapidly growing user base and the growth of the Internet, the average expertise of system administrators has decreased. Overworked administrators cannot afford to maintain and configure many packages manually.

Converting OSS packages to use GNU autotools—Autoconf [5], Automake [6], and Libtool [7]—addresses the aforementioned problems in five ways:

1. **Standard tests**: Autoconf has a large set of standard portable tests that were developed from practical experiences of the maintainers of several GNU packages. Autoconf tests for features by actually exercising those features (e.g., compiling and running programs that use those features). Packages that use Autoconf tests are automatically portable to all of the platforms on which these tests work.
2. **Consistent names**: Autoconf produces uniform macro names that are based on features. For example, code which uses Autoconf can test if the system supports a reliable `memcmp` function using `#ifdef HAS_MEMCMP`, rather than depending on system-specific macros (e.g., `#ifndef SUNOS4`). Autoconf provides a single macro per feature, reducing the need for complex or nested macro expressions. This improves code readability and maintainability.
3. **Shared libraries**: By using Libtool and Automake along with Autoconf, a package can build shared or static libraries easily, removing a lot of custom code from sources and makefiles.
4. **Human factors**: Building packages that use autotools is easy. Administrators are becoming increasingly familiar with the process and the standard set of features autotools provide (i.e., run `./configure` and then `make`). Administrators do not need

to configure the software package manually prior to compilation and they are likely to make fewer mistakes. This standardization speeds up installation and configuration of software.

5. **Extensibility**: Finally, software maintainers can extend Autoconf by writing more tests for specific needs. For example, we wrote specific tests for the Am-utils package that detect its interaction with certain kernels. This allowed us to separate the common code from the more difficult-to-maintain platform-specific code.

Our experiences with maintaining the Amd package clearly show the benefits of autotools. When we converted the Amd package [9, 12] to use autotools, the code size was reduced by more than one-third and the code became clearer and easier to maintain. Fixing bugs and adding new features became easier and faster, even major features that affected significant portions of the code: NFSv3 [8] support, Autofs [1] support, and a run-time automounter configuration file `/etc/amd.conf`. New features that we added immediately worked on many supported systems and bugs fixes did not introduce additional bugs.

3 Autotooling

The basic idea behind Autoconf is to determine a feature's availability by running a small test that actually uses the feature. The test often writes a small program on the fly, compiles it, and possibly links and runs it. This is a reliable method of detecting features. Autoconf-generated configuration scripts are portable. When run, they use portable shell scripting and tools such as `sed` and `grep`. Building `configure` scripts requires GNU M4, but only by the maintainers of the package, not by those building the package.

In this section we explain how GNU autotools work and detail the types of autotool tests we used or wrote for use with Am-utils.

3.1 Autoconf

An Autoconf `configure` script is built from a `configure.in` file that contains a set of Autoconf M4 macros to use. Autoconf translates the M4 macros into their respective portable shell code. For example, to test if the system supports the `bzero` function, we used this M4 macro: `AC_CHECK_FUNCS(bzero)`. Autoconf translates this M4 macro call into shell code that creates, compiles, and links the following program:

```
#include "confdefs.h"
char bzero(); /* forward definition */
int main() {
  bzero();
  return 0;
}
```

If the program compiles and links successfully, the `configure` script defines a CPP macro named `HAVE_BZERO` in an automatically-generated configuration file named `config.h`. This macro is based on the existence of the feature and can be used reliably in the sources for the package. Note that it is not necessary to run this program to determine if `bzero` exists. In fact, the above program will fail to run properly because the `bzero` function was not given proper parameters.

Assuming the package also checks for the existence of the `memset` function, the maintainers of the package can use the following portable code snippet reliably:

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif /* HAVE_CONFIG_H */

#ifndef HAVE_BZERO
# ifdef HAVE_MEMSET
#  define bzero(ptr, len) \
          memset((ptr), 0, (len))
# else /* not HAVE_MEMSET */
#  error neither bzero nor memset found
# endif /* not HAVE_MEMSET */
#endif /* not HAVE_BZERO */
...
   struct nfs_args na;
   bzero(&na, sizeof(struct nfs_args));
```

There are four categories of feature tests that we used or developed during the autotooling process for Am-utils:

1. No Autoconf test needed.
2. Simple existing Autoconf tests were available.
3. New simple Autoconf tests had to be written.
4. Static Autoconf tests had to be written.

3.1.1 No Test Needed

These features are common to most Unix platforms and they are easily available from standard system header files. No Autoconf tests or macros were required; the user need only include the correct header files. For example, to detach from its controlling terminal, one of the methods a long-running daemon such as `amd` uses is the `TIOCNOTTY ioctl`. By simply including the right header files, we could write C code that performed an action only if `TIOCNOTTY` was defined.

3.1.2 Simple Existing Tests

These are Autoconf macros for which an existing test was suitable. Over 70% of the Autoconf tests used by Am-utils and other large packages fall into this category. A few examples are:

- `AC_CHECK_LIB(rpcsvc, xdr_fhandle)` checks if the `rpcsvc` library includes the `xdr_fhandle` function. If so, the test ensures that `-lrpcsvc` is used when linking binaries.
- `AC_REPLACE_FUNCS(strdup)` tests if the `strdup` function exists in any standard library, and if not, adds `strdup.o` to the objects to build. The package maintainers are then required to supply a replacement `strdup.c` file.
- `AC_CHECK_HEADERS(nfs/proto.h)` checks if that header file exists. If so, the test defines the macro `HAVE_NFS_PROTO_H`, which can be used in the code to include the header file only if it exists.
- `AC_CHECK_TYPE(time_t, unsigned long)` looks for a definition of `time_t` in standard header files such as `<sys/types.h>`. If not found, the macro defines the type to `unsigned long`. Portable code need only use `time_t`.
- `AC_FUNC_MEMCMP` tests if the `memcmp` library function exists and if is 8-bit clean; some versions of this function incorrectly compare 8-bit data. This is an example of how Autoconf can help to detect bugs in system software and offer the package maintainer a workaround option.

Interestingly, the full set of all Autoconf tests also serves as a testament to the total sum of operating system variability. The macros list numerous features that possibly differ from system to system.

3.1.3 Newly Written Tests

These were macros for which no existing Autoconf test existed and thus had to be written. The key here is to design and write tests that could *reliably* determine a feature all of the time, regardless of the tools used.

We describe only one example in this paper: `AC_CHECK_FIELD`, a test to determine if a given structure contains a certain field.[1] This test can be used to handle structures with the same name that have different members across different platforms. Our M4 test macro is used as follows:

```
AC_CHECK_FIELD(struct sockaddr, sa_len)
```

The macro takes two arguments: the first (`$1`) is the name of the structure and the second (`$2`) is the name of the field. The macro creates and tries to compile the following small program:

```
main() {
  $1 a;
  char *cp = (char *) &(a.$2);
}
```

If the above program compiles successfully, the test defines a CPP symbol whose name is automatically constructed: `HAVE_FIELD_STRUCT_SOCKADDR_SA_LEN`. Code that uses this CPP symbol can perform the proper actions when the `sockaddr` structure contains an `sa_len` field.

3.1.4 Static Autoconf Tests

Autoconf tests must be 100% deterministic. We identified several classes of problems where a simple reliable test could not be written. We wrote these tests statically: the result of the test is hard-coded based on the operating system name and version. These include tests for certain types, for kernel features, for system bugs, and for other features which would have required complex tests.

1. **Types**: The types of arguments passed to functions or used in structure fields cannot be detected easily. C is not a type-safe language. Some C compilers do not always fail when a type mismatch occurs, even with special compiler flags. One example of such a test is determining the type of the NFS file handle field in `struct nfs_args`: it can be a fixed-size buffer, one of several other structures, or a pointer to any of those.
2. **Kernel features**: Kernel internals are difficult to probe, even with root access. For example, determining how the `mount(2)` system call works is not practical because it requires mounting a known file system and superuser privileges. Worse, if the resource being mounted is a remote file system (e.g., NFS), the client also needs to know the name of the server exporting that file system and the name of the exported path on the server.
3. **System bugs**: Some systems have bugs that cannot be easily detected. For example, all versions of Irix up to 6.4 use an NIS function `yp_all` that leaks a TCP file descriptor opened to the bound NIS server `ypserv`. To detect this bug, a program must run on a configured NIS system and know which NIS maps to download—information that is specific to the site. We considered and rejected several more complex alternatives to detect this bug; it was simpler to hard-code the answer.

4. **Complex tests**: If the Autoconf test being written is too complex or long, generally more than half a page of M4, C, and sh code, and is used only once or twice throughout the configure script, it is better to write it statically. A shorter test, even a static one, is often more readable and helps to reduce the maintenance effort needed for the package.

The following example illustrates when a static test was preferable. Automounters include an entry in mount tables (e.g., `/etc/mtab`) that is set as "hidden" from the `df(1)` program because there is not much meaningful data that `statfs(2)` can return to `df` for that mount point. Amd uses the "ignore" or "auto" mount type to tell the system to hide that mount entry from `df`. We found that this feature was difficult to test automatically: some systems do not define these mount types in their header files but still use them (hard-coded in the vendor's own tools). Other systems define both, but prefer one over the other. A few systems define one or more of them but use an entirely different mechanism to hide mount entries: a mount table flag. Lastly, some systems do not have the ability to hide mount entries. We wrote the test simply and statically as follows:

```
case "${host_os}" in
  irix* | hpux10* )
    ac_cv_hide_mount_type="ignore"  ;;
  sunos4* )
    ac_cv_hide_mount_type="auto"    ;;
  * )
    ac_cv_hide_mount_type="nfs"     ;;
esac
```

3.2 Automake

Automake [6] can automatically generate any number of Makefile templates for use when configuring packages. These Makefile templates contain the exact definitions for compiling, linking, installing programs and auxiliary files, and more. Automake allows for many features tested during the process of configuring the package to be passed on to Makefile templates. These Makefile templates are used by the `configure` script at configure time; the script performs simple variable substitution on the templates, to produce the actual `Makefiles` used to compile the package. The latter are the final Makefiles for that specifically-configured package and include any additional site overrides.

One example of Automake's usefulness is that it creates templates that contain generic definitions for the libraries that applications need to link with. These libraries are detected early in the configuration process. This way the exact set of libraries needed is used during build time. The names and locations of these libraries do not have to be statically configured or specified by the user.

One disadvantage of Automake is that it produces long and complex Makefile templates. These are more difficult to debug and understand because of their length and the large number of Makefile features they use which are intended to assist `configure` in producing a final Makefile.

To use Automake, a package maintainer writes small `Makefile.am` Automake template files that define the bare minimum that needs to be known at that point; this is often just the names of target files and the sources used to produce those targets. Then, the maintainer calls a small number of special M4 macros in their `configure.in` file which tell Autoconf that this package uses Automake-generated Makefiles. Next, the maintainer runs `automake` to produce the Makefile template files named `Makefile.in`. The `configure` script reads `Makefile.in` files at configure time and generates the final `Makefiles` used to compile the package.

3.3 Libtool

Libtool [7] automates the building, linking, and installation of shared or static libraries. Shared library support is specific to a given system. Different compiler, linker, and assembler options are often used to build shared libraries and those options depend on the specific development tools used. Some shared libraries use different extensions such as `.so` or `.sl`. Different systems use different rules for versioning of shared libraries. Some other systems require setting environment variables such as `$LD_LIBRARY_PATH` in order to run a binary with the proper shared library.

To use Libtool, a package maintainer calls a small number of special M4 macros in their `configure.in` file. Also, the maintainer must use a slightly different way of specifying libraries in various `Makefile.am` files, to indicate to Autoconf and Automake that this package will use Libtool to support both shared and static libraries.

4 Evaluation

When a package uses autotools such as Autoconf, it generally becomes easier to maintain. However, even Autoconf has its limitations. The first goal of this section is to provide a method of evaluating a package's complexity for developers who are using or considering using autotools for that package. Note that we are specifically concerned with complexity concerning the portability of that package to newer operating environments. The second goal of this section is to show the benefits and limitations of using autotools.

Table 1 lists the packages that we evaluated. We picked ten large, popular packages that use autotools, including Am-utils. We also evaluate the Amd package, which is Am-utils before it was autotooled. We evaluated as many versions of these packages as we could find, spanning development cycles of 2 to 9 years. This ensures that our reported results are sufficiently stable, given a large number of versions spanning several years.

Package	Versions	Year-span
amd	10	2.6
am-utils	48	4.7
bash	9	4.6
bin-utils	9	4.9
emacs	12	5.1
gcc	11	9.1
gdb	4	4.1
glibc	11	5.2
openssh	58	2.0
tcl	38	8.7
tk	37	8.1

Table 1: The packages evaluated in this section, the number of versions of each package we evaluated, and the overall span of release years for those versions.

Figures 1 through 6, include "error" bars showing one standard deviation off of the mean. Since we have evaluated a number of packages and versions for each, the standard deviation accounts for the general variation in size and complexity of the package over time.

4.1 Code Complexity

A typical metric of code complexity is a count of the number of lines of code in the package, as can be seen in Figure 1. As we can see, the four largest packages are Binutils, Gcc, Gdb, and Glibc. The number of lines of code in a package is one useful measure of the effort involved in developing and maintaining the package, but may not tell the whole story.

A different measure of the portability complexity of a package is the number of CPP conditionals that appear in the code: `#if`, `#ifdef`, `#ifndef`, `#else`, and `#elif`. Each of those statements indicates one extra code compilation diversion. Generally, each of those CPP conditionals account for some difference between systems, to ensure that the code can compile cleanly on each system. In other words, the effort to port a software package to multiple systems and maintain it on those is related to the number of CPP conditionals used.

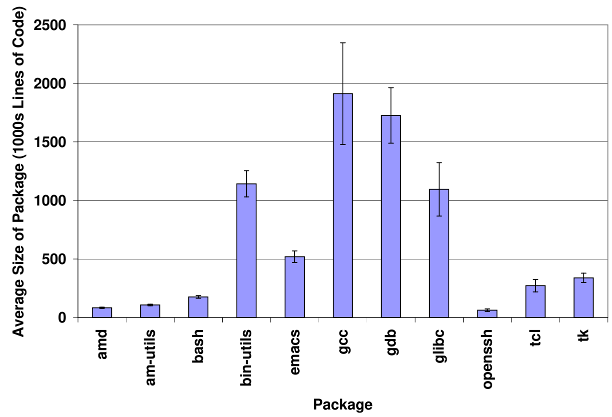

Figure 1: Average size of packages in thousands of lines of code.

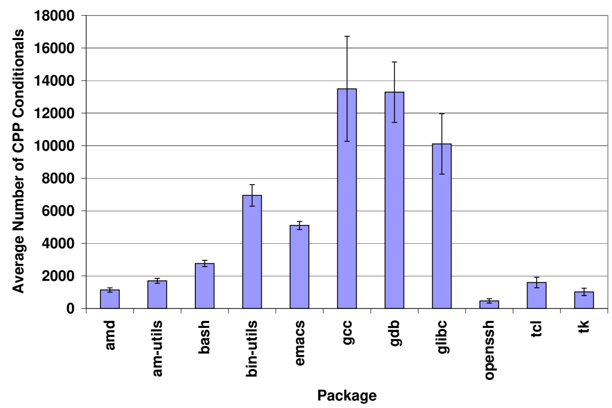

Figure 2: Average number of CPP conditionals per package.

Figure 2 shows the average number of CPP conditionals per package. Since Autoconf supports conditional whole source file compilations, we counted each of those conditionally-compiled source files as one additional CPP conditional. Here, the same packages that have the most lines of code (Figure 1) also have the most number of CPP conditionals. This is not surprising: as the size of the package grows, so the number of CPP conditionals is likely to grow. This suggests that perhaps neither code size nor absolute number of CPP conditionals provide a good measure of code complexity.

Next, we combined the above two metrics to provide a normalized metric of code complexity for the purposes of portability and maintainabilty of code over multiple operating systems. In Figure 3 we show the average number of CPP conditionals per 1000 lines of code. We notice three things in Figure 3.

First, the difference between the most and least com-

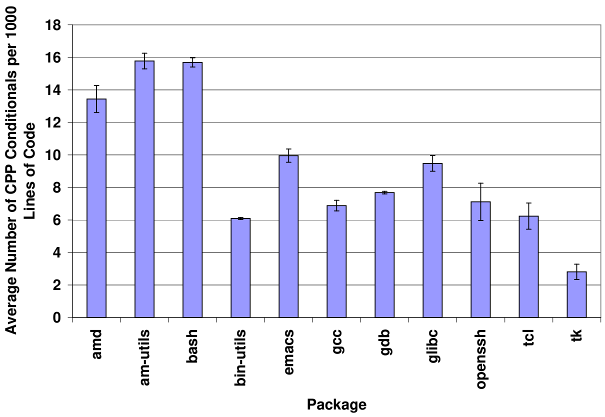

Figure 3: Average number of CPP conditionals per 1000 lines of code.

plex packages is not as large as in Figures 1 and 2. In those two, the difference was as much as an order of magnitude, whereas in Figure 3 the difference is a factor of 2–4.

Second, the standard deviation in this figure is smaller than in the first two. This means that although larger packages do have more CPP conditionals, the average distribution of CPP conditionals is almost fixed for a given package. The implication of this is also that a given package may have a native (portability) complexity that is not likely to change much over time and that this measure is related to the nature of the package, not its size.

The third and most important thing we notice in Figure 3 is that the packages that appear to be very complex in Figures 1 and 2 are no longer the most complex; leading in average distribution of CPP conditionals are Am-utils and Bash. To understand why, we have to understand what makes a software package more complex to port to another operating system. For the most part, languages such as C and C++ are portable across most systems. The biggest differences come when a C program begins interacting with the operating system and the system libraries, primarily through system calls. Although POSIX provides a common set of system call APIs [3], not all systems are POSIX compliant and every system includes many additional system calls and `ioctls` that are not standardized. Furthermore, although the C library (`libc`) provides a common set of functions, many functions in it and in other libraries are not standardized. For example, there are no widely-standardized methods for accessing configuration files that often reside in `/etc`. Similarly, software (e.g., `libbfd` in Binutils) that handles different binary formats (ELF, COFF, a.out) must function properly across many platforms regardless of the binary formats supported by those platforms.

Despite their large size, Binutils, Gcc, Gdb, and Glibc—on average—do not use as many operating system features as Bash and Am-utils do. For example, Gcc and Binutils primarily need to be able to read files, process them internally (parsing, linking, etc.), and then write output files. Most of their complexity exists in portable C code that performs file parsing and target format generation. Whereas Gcc and Binutils are large and complex packages in their own right, *porting* them to other operating systems may not be as difficult a job as for a program that uses a wide variety of system calls or a program that interacts more closely with other parts of the running system. (In this paper we do not account separately for package-specific portability complexities: Gcc and Binutils to new architectures, Am-utils to new file systems, Emacs and Tk to new windowing systems, etc.)

For example, Bash must perform complex process and terminal management, and it invokes many system calls as well as special-purpose `ioctls`. Amd (part of Am-utils) is a user-level file server and interacts heavily with the rest of the operating system to manage other file systems: it interacts with many local and remote services (NFS, NIS/NIS+, DNS, LDAP, Hesiod); it understands custom file systems (e.g., loop-device mounts in Linux, Cachefs on Solaris, XFS on IRIX, Autofs, and many more); it is both an NFS client and a local NFS server; and it communicates with the local host's kernel using an asynchronous RPC engine. By all rights, an automounter such as Amd is a file system server and *should* reside in the kernel. Indeed, Autofs [1, 4] is an effort to move the critical parts of the automounter into the kernel.

4.2 Autoconf Tests

Before building a package, it must be configured. The number of Autoconf (and Automake and Libtool) tests that a package must perform is another useful measure of the complexity of the package. The more tests performed, the more complex the package is to port to another system, since the package requires a larger number of system-discriminating features.

Figure 4 shows the average number of tests that each package performs. The figure validates some of what we already knew: that Binutils, Gcc, Gdb are large and complex. But we also see that Am-utils performs more than 600 tests: only Gdb performs more tests on average. This confirms that Am-utils is indeed a complex package to port, even though its size is more than ten times smaller than Gcc or Gdb.

Since Autoconf comes with many useful tests already, we also measured how many new Autoconf tests the

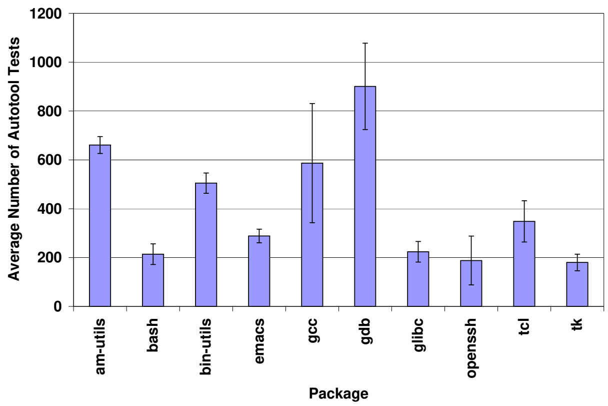

Figure 4: Average number of autotool tests (Autoconf, Automake, and Libtool M4 tests). The large standard deviation for Gcc is due to the fact that Gcc 2.x used a few canned configurations, whereas Gcc 3.x began using many Autoconf tests. Amd is excluded because it is the Am-utils package before autotooling, and hence includes no Autoconf tests.

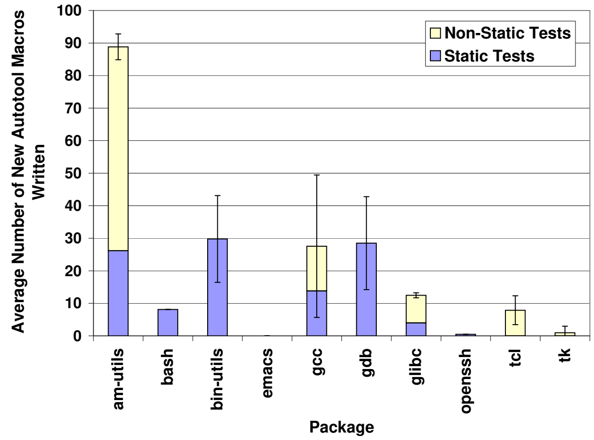

Figure 5: Average number of new Autoconf tests written and the portion of those that are static tests. Amd is excluded because it is the Am-utils package before autotooling, and hence includes no Autoconf tests.

package's maintainers wrote. The number of new macros that are written shows two things. First, that Autoconf is limited to what it already supports and that new macros are almost always needed for large packages. Second, that the number of new macros needed indicates that a given package may be more complex.

Figure 5 shows the average number of new M4 macros (Autoconf tests) that were written for each package. As we see, most packages do not need more than 10 new tests, if any. Binutils, Gcc, and Gdb are more complex and required about 30 new tests each. Am-utils, on the other hand, required nearly 90 new tests. According to this metric, Am-utils is more complex than the other packages we measured because it requires more tests that Autoconf does not provide. Indeed this has been true in our experience developing and maintaining Am-utils: many tests we wrote try to detect kernel features and internal behavior of certain system calls that none of the other packages deal with (for example, how to pass file-system–specific mount options to the `mount(2)` system call).

Figure 5 also shows the portion of static macros that were written. As we described in Section 3.1.4, these are macros that cannot detect a feature 100% deterministically and are often written as a case statement for different operating systems. In other words, these tests cannot use the power of Autoconf to perform automated feature detection. As we see in Figure 5, most packages need a few such static macros, if any. Again, Am-utils takes the lead on such static macros. The main reason for this is that a reliable way to test such features is impossible without superuser privileges and knowledge of the entire site (i.e., the names, IP addresses, and exported resources of various network servers).

The conclusion we draw in this section is that although Autoconf continues to evolve and provide more tests, maintainers of large and complex packages may still have to write 10–30 custom macros. Moreover, some packages will always need a number of static macros, for those features that Autoconf cannot test in a reliable, automated way.

4.3 Amd and Am-utils

In Sections 4.1 and 4.2 we established that Am-utils is a more complex package to port than it appears from looking purely at its size. In several ways, Am-utils represents an upper bound for the complexity of portable C code: of the packages examined it has the most dense distribution of CPP conditionals and the largest number of custom macros. In addition, it performs critical file system services that are often part of the kernel proper. Therefore, analyzing Am-utils in more detail provides more insight into the process of maintaining and porting large or complex packages. Moreover, since we have maintained this code for nearly ten years, we can provide a unique perspective on the history of the development of Am-utils dating back to well before it used autotools.

In Figure 6 we see the code size for all released versions of Am-utils. The vertical dashed line separates the autotooled code on the right from the non-autotooled one on the left (before autotooling, the package had a "upl" versioning scheme). The most important factor is the drop in code size after autotooling. The last version before autotooling (amd-upl102) was 91640 lines long.

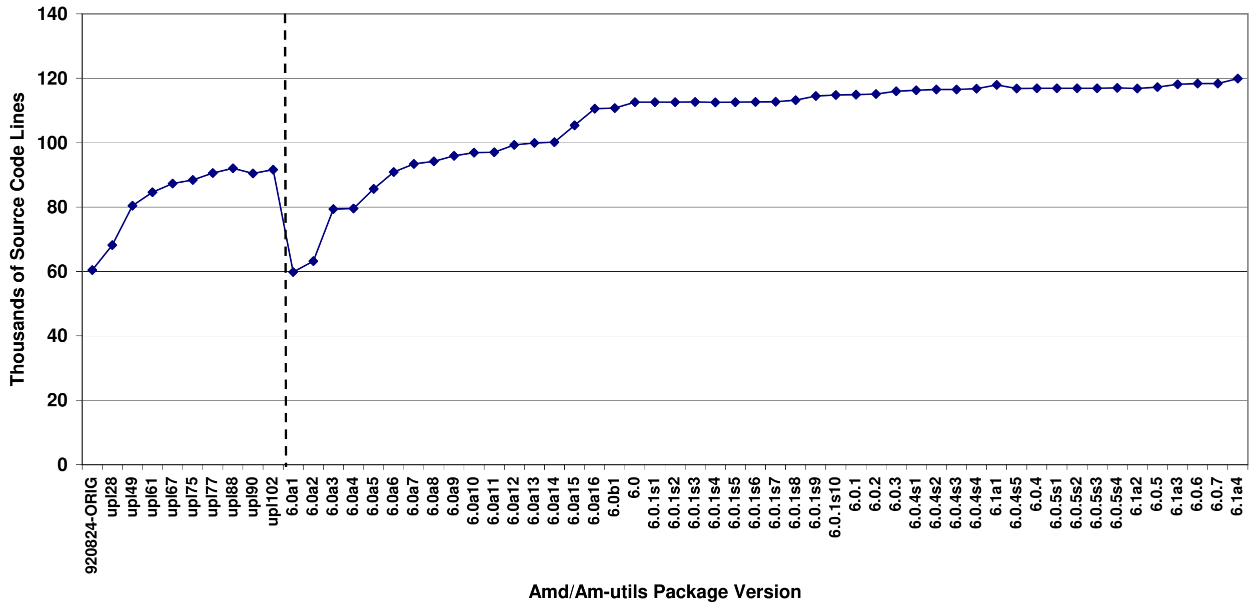

Figure 6: Code size of all Amd and Am-utils package versions. Versions to the right of the vertical line were autotooled.

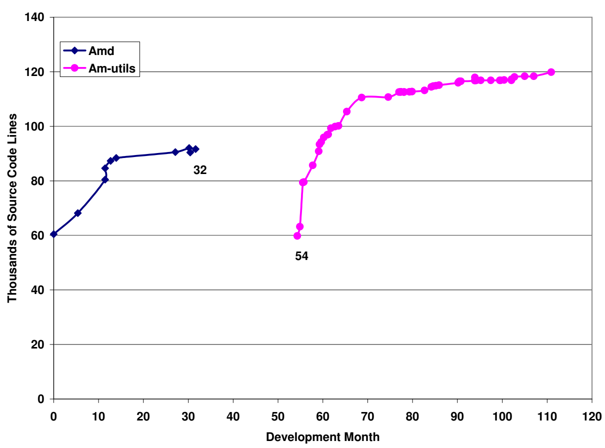

Figure 7: Development timeline of Amd and Am-utils, spanning nearly 10 years. There is a dramatic increase in code size (and hence features) for Am-utils in the first year after the package had been autotooled.

The first version after autotooling (am-utils-6.0a1) was only 59804 lines long. The two versions offered identical features and behavior, but the latter used Autoconf, Automake, and Libtool. This drop of 35% in code size was afforded thanks to the code cleanup and simplification that resulted from using the GNU autotools.

Since the release of many versions occurred over a period of nearly ten years, we show in Figure 7 the timeline for each release and the code size since the first release. We can see that for the first 32 months, nearly three years, the non-autotooled package continued to grow in size. However, that growth in size was accompanied by a serious increase in difficulty to maintain the package which hindered the package's evolution; the older code contained a large number of multi-level nested CPP macros, often resulting in what is dubbed "spaghetti code." It took nearly two more years before the first autotooled version of Amd was released. Of those 22 months, about 6 months prior to month 54 were spent learning how to use Autoconf, Automake, and Libtool, understanding the inner working of Amd, and adapting the code to use autotools.[2]

The autotooling effort paid off significantly in two ways. First, the autotooled version was more than one-third smaller. Second, the autotooling process allowed us to add new features to Amd and port it to new systems with minimal effort. For the first 12 months after its initial autotooled release, Am-utils grew by 70%, adding major features such NFSv3 [8] support, Autofs [1] support, a run-time automounter configuration file `/etc/amd.conf`, and offering dozens of new ports. This growth in size did not complicate the code much, as can be seen from the small standard deviation for Am-utils in Figure 3. The growth rate has reduced over the past two years, as the Am-utils package became more stable and fewer new features or ports were required.

The conclusion we draw from our experiences is that large packages can benefit greatly from using autotools such as Autoconf. Autoconf-based packages are easier to maintain, develop, and port to new systems.

4.4 Performance

Autotooling a package bears a build-time performance cost. Before compiling a package, `configure` must run to detect system features. This detection process can consume a lot of time and CPU resources. Therefore, Autoconf supports caching the results of a `configure` run, to be used in later invocations. These cached results can be used as long as no changes are made to the system that could invalidate the cache, such as an OS upgrade, installation of new software packages, or de-installation of existing software packages.

We measured the elapsed time it took to build a package before and immediately after it was autotooled. We used the last version of Amd before it was autotooled (upl102) and the first version after it was autotooled (6.0a1) because these two included functionally identical code. We ran tests on a number of different Unix systems configured on identical hardware: a Pentium-III 650MHz PC with 128MB of memory. We ran each test ten times and averaged the results. The standard deviations for these tests were less than 3% of the mean.

Action and Package	**Time**
Build Amd-upl102	35.4
Configure Am-utils-6.0a1 (no cache)	102.6
Configure Am-utils-6.0a1 (with cache)	24.2
Compile Am-utils-6.0a1	73.1

Table 2: Time (seconds) to Configure and Build Amd Packages

Table 2 shows the results of our tests. We see that just compiling the newly autotooled code takes more than twice as long. That is because the autotooled code includes long automatically generated header files such as `config.h` in every source file—despite the fact that the autotooling process reduced the size of the package itself by one-third. Worse, configuring the newer Am-utils package alone now takes more than 100 seconds. However, after that first run, re-configuring the package with cached results runs four times faster.

If we consider the worst-case overall time it takes to build this package, including the configuration part without a cache, then building Am-utils-6.0a1 is nearly five times slower than its functionally-equivalent predecessor, Amd-upl102.

4.5 Autotool Limitations

Through this work, we identified five limitations to GNU autotools. First, developers must be fairly knowledgeable in using these autotools, including understanding how they work internally.

Second, building code that was autotooled often takes longer than non-autotooled equivalents. However, given ever-increasing CPU speeds, this limitation is often not as important as ease of maintenance and configuration.

Third, developing code with autotools requires using a GNU version of the M4 processor. Also, Autoconf depends on the native system to provide stable and working versions of the Bourne shell `sh`, as well as `sed`, `cpp`, and `egrep` among others. Even though such tools come with most Unix systems, they do not always behave the same. When they behave differently, maintainers of GNU autotools must use common features that will work portably across all known systems.

Fourth, although autotools support cross-compilation environments which further helps the portability of cross-compiled code, Autoconf generally cannot execute binary tests meant for one platform on another. This limits Autoconf's ability to detect certain tests that require the execution of binaries.

Last, developers may still have to write custom tests and M4 macros for complex or large packages. Developing, testing, and debugging such tests is often difficult since they intermix M4 and shell syntax.

5 Conclusions

The main contribution of this paper is in quantifying the benefits of autotools such as GNU Autoconf: showing how much they help the portability of large software packages and illustrating their limitations.

We also showed several useful metrics for measuring the complexity of code when it comes to its portability: the average distribution of CPP conditionals in the code, the number of new Autoconf tests that had to be written, and the portion of those tests that were static and therefore beyond the normal capabilities of Autoconf. Using these metrics we showed how packages with more lines of code may not be as difficult to port as packages that use a more diverse set of system features.

In analyzing the evolution of the Am-utils package, we showed how the package benefited from autotooling: its code size was reduced dramatically and the code base was made cleaner, thus allowing rapid progress on new feature implementation.

The GNU autotools we used also have limitations. First, maintainers must become intimately familiar with the autotools. Second, building autotooled packages takes longer. Third, even autotools cannot solve all portability problems; large-package maintainers usually have to write their own custom tests. Fourth, certain tests cannot be executed in a cross-compiled environment. Nevertheless, in the long run, once an initial autotooling effort has taken place, an autotooled package is easier to maintain on existing systems and port to new or diverging systems.

Although a count of code lines had been used for years as a metric of code complexity, we believe that metric oversimplifies the process of portable software development. In the future we would like to automate the process of quantifying the complexity of a package. We plan on building tools that will parse autotool files and related C code, separately evaluating conditionally-included code from unconditionally-included code. This will allow us to evaluate how much of a given autotooled package is highly portable code and how much code is densely populated with system-specific code. In addition, we would like to account for package-specific portability complexities (i.e., how new architectures affect Gcc and Binutils, new file systems affect Amd, and new windowing systems affect Emacs and Tk).

6 Acknowledgments

We would like to thank the many members of the Am-utils developer's list and our co-maintainers for their contributions to the code, recommendations, and comments. Many of them provided us remote and superuser access to different systems so we could compile and test new code on as many platforms as possible. Several people made significant contributions to the code: Ion Bădulescu implemented Autofs support and maintains the Linux support in Am-utils, despite the increase in divergence among Linux systems; Randall S. Winchester and Christos Zoulas contributed the initial NFS V.3 code.

Additional thanks go to the anonymous Usenix reviewers and especially Chuck Cranor, Chris Demetriou, and Niels Provos for their valuable comments. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374, as well as by an HP/Intel gift number 87128.

To retrieve the Am-utils software, including documentation and the dozens of new M4 macros written for Am-utils, visit *http://www.am-utils.org*.

References

[1] B. Callaghan and S. Singh. The autofs automounter. In *Proceedings of the Summer USENIX Technical Conference*, pages 59–68, Summer 1993.

[2] J. S. Haemer. Imake rhymes with mistake. *;login:*, 19(1):32–3, Jan-Feb 1994.

[3] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Technical Report STD-1003.1, ISO/IEC, 1996.

[4] R. Labiaga. Enhancements to the Autofs Automounter. In *Proceedings of the Thirteenth USENIX Systems Administration Conference (LISA '99)*, pages 165–174, Seattle, WA, November 1999.

[5] D. MacKenzie. Autoconf: Creating automatic configuration scripts. Technical report, FSF, 1996.

[6] D. MacKenzie and T. Tromey. Gnu automake. Technical report, FSF, 1997.

[7] G. Matzigkeit. Gnu libtool. Technical report, FSF, 1997.

[8] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, June 1994.

[9] J. S. Pendry and N. Williams. *Amd – The 4.4 BSD Automounter*, 5.3 alpha edition, March 1991.

[10] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, October 2000.

[11] L. Wall, H. Stenn, and R. Manfredi. dist-3.0. Technical report, 1997. ftp.funet.fi/pub/languages/perl/CPAN/authors/id/RAM/.

[12] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.

Notes

[1] A similar test to detect names of fields within structures was recently added to Autoconf, partially based on our efforts in writing and contributing M4 test macros to the OSS community.

[2] Amd was originally written by Jan-Simon Pendry and Nick Williams, not the authors of this paper. Therefore we did not initially understand every part of the original 60432-line code base.