Feature-Based Portability

Glenn S. Fowler (gsf@research.att.com)
David G. Korn (dgk@research.att.com)
John J. Snyder (jjs@research.att.com)
Kiem-Phong Vo (kpv@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Current computing platforms encompass a dizzying variety of hardware and software. A software application may live or die based on how portable it is. Much has been written and talked about how to enhance portability. But few tools are available to support writing portable code and, more importantly, to encode porting knowledge. This paper describes IFFE, a tool and an accompanying programming style that supports software portability. IFFE has enabled the porting and construction of many large software applications on heterogeneous platforms with virtually no user intervention.

1. Introduction

Over the past 10 years our department at AT&T Bell Laboratories has been engaging in writing a number of popular software tools and libraries. Some examples are KSH [BK89], a shell language, NMAKE [Fow85], a language and system to build running code from source, EASEL [Vo90, FSV94], a language and system to build end-user applications, and *sfio* [KV91], a library for buffered I/O. These tools and libraries critically depend on various resources provided in the underlying platforms. A major problem is that such platform resources are not always available or usable in the same form. For example, to tell whether or not a file descriptor is ready for I/O, on a BSD-derived system, one should use the `select()` system call while on newer System V systems, `poll()` is required. The problem is exacerbated by the fact that there are many hybrid systems, some from the same vendor, that provide mixed services. In a different direction, most systems come with standard libraries such as string and mathematical packages but their implementations vary in quality. An extreme case is the VAX family of machines that come with hardware instructions for certain string and character look-up operations that are more efficient than any handcrafted software. It is desirable to take advantage of such platform-specific features to optimize the software. Of course, in all cases, we have to be certain that a platform feature used will work as expected. In sum, the porting problem is this: how can we certify that a particular feature exists on a particular software/hardware platform and that it does what is required?

This paper describes a tool IFFE (IF Features Exist) and an accompanying programming style to help with writing portable code and gives a brief comparison of IFFE to other approaches. IFFE has enabled us to: (1) port software to new platforms with minimal changes, (2) codify learned knowledge during porting, (3) apply such knowledge without relying on users to specify software/hardware parameters at each installation, and, last but not least, (4) take advantages of special platform features to tune for performance.

2. A programming style for portability

Our overall approach to portability is to program applications against high level libraries that hide differences among underlying platforms. Then porting effort is mostly confined to the library code. The traditional approach for selecting different code variants is to use "`#ifdef` *selector*" where *selector* is a predetermined symbol based on some broad categorization of machine type (e.g., `sun` or `sgi`) or operating system type (e.g., `BSD` or `SYSV`). Such a broad categorization is convenient and does work in limited cases. It is also necessary because the specific value of *selector* is typically supplied by some user during a build and most users neither know nor have the means to find out and evaluate alternatives in the full set of locally available features. However, in modern environments where mixtures of services are typical, more often than not this traditional way of code selection will miss the mark and lead to the construction of bad code.

We solve the porting problem by applying a programming style supported by IFFE. It is best to show this with an example. Consider the following code fragment taken from the source of the `sfpopen()` function of the *sfio* library:

```
1:  #include     "FEATURE/vfork"
2:  #if _lib_vfork
3:  #   define fork vfork
4:  #   if _hdr_vfork
5:  #       include     <vfork.h>
6:  #   endif
7:  #endif
```

Line 1 includes a file `FEATURE/vfork` that defines two symbols, `_lib_vfork` and `_hdr_vfork`. Line 2 tests `_lib_vfork` for the existence of the system call `vfork()`. Line 3 redefines `fork()` to `vfork()` if it exists. `sfopen()` uses `fork()` to create a child process which, after some minor processing, will be overlaid by a new command. So redefining `fork()` as `vfork()` is good as the latter does the same job without the expensive operation of copying all data of the parent. This use of `vfork()` works fine except on a SUN SPARC which has a major problem that registers modified by a child process get propagated back to its parent. SUN provides a compiler directive in the header file `vfork.h` to generate code that avoids this bug. The existence of this file is tested on line 4 and its inclusion is done on line 5. It is worth emphasizing that even though this problem is currently known to occur only on a SUN SPARC, its solution is targeted at the nature of the problem and not at the machine. Further, the solution is encoded in a form independent of machine architectures.

To complete the example, we need to see how the file `FEATURE/vfork` containing the symbols `_lib_vfork` and `_hdr_vfork` can be correctly and automatically generated. This is done via the IFFE language and system. The keen reader may have noticed that a subdirectory `FEATURE` is used to store the header file `vfork` which contains the definitions of the required tokens. This file is generated from a specification file `vfork` in a parallel directory `features`. The content of `features/vfork` is:

```
lib vfork
hdr vfork
```

The line "`lib vfork`" determines if `vfork()` is a function in some standard library (e.g., `/lib/libc.a`) by generating, compiling and linking a small test program that contains a `vfork()` call. Similarly, the line "`hdr vfork`" determines if the header file `vfork.h` exists by compiling a small program containing the line "`#include <vfork.h>`". Below is the output file `FEATURE/vfork` for a SUN SPARC. Note that to prevent errors with multiply inclusions the generated symbols are automatically wrapped with the wrapper `#ifndef` and `#endif` which is generated from the base name of the feature test file `features/vfork` and `sfio`, the parent directory of `features` and package name. By the way, in case the reader may wonder why `FEATURE` does not have an `S` to parallel `features`, this is done to distinguish the two directories on operating systems such as Windows NT where cases are indistinguishable in directory and file names.

```
#ifndef _def_vfork_sfio
#define _def_vfork   1
#define _lib_vfork   1   /* vfork() in default lib(s) */
#define _hdr_vfork   1   /* #include <vfork.h> ok     */
#endif
```

Though the above example works, we are actually a little too trusting as compilability is not equivalent to execution correctness. For complete safety, IFFE scripts can specify programs that must compile, link and execute successfully. Below is another IFFE specification from *sfio* that tests for the correct register layout of a given VAX compiler:

```
vax asm note{ standard vax register layout }end execute{
    main()
    {
    #ifndef vax
        return absurd = 1;
    #else
        register int    r11, r10, r9;
        if(sizeof(int) != sizeof(char*))
            return 1;
        r11 = r10 = r9 = -1;
        asm("clrw    r11");
        if(r11 != 0 || r10 != -1 || r9 != -1)
            return 1;
        asm("clrw    r10");
        if(r11 != 0 || r10 != 0 || r9 != -1)
            return 1;
        asm("clrw    r9");
        if(r11 != 0 || r10 != 0 || r9 != 0)
            return 1;
        return 0;
    #endif
    }
}end
```

The above code will compile and run correctly only on a VAX with a proper compiler. If that is the case, the output would be as below and we would know that the register layout is as expected so that certain hardware instructions can be used safely for optimization.

```
#define _vax_asm 1   /* standard vax register layout */
```

IFFE specifications can be integrated with `makefile`s in the obvious fashion. Users of the NMAKE system for code construction enjoy this integration automatically since NMAKE scans the source code for any implicit header file prerequisites (a.k.a. `#include` dependencies) including the `FEATURE` files. The additional NMAKE metarule shown below provides the action to generate the `FEATURE` files. Note that where old MAKE is still used, NMAKE can also be used to generate `makefile`s that contain all such header dependencies.

```
FEATURE/% : features/% .SCAN.c (IFFE) (IFFEFLAGS)
    $(IFFE) $(IFFEFLAGS) run $(>)
```

To summarize, the programming style that we adhere to is:

1. Determine needed features that may have platform specific implementations.
2. Write IFFE probes to determine the availability and correctness of such features.
3. Instrument `makefile`s to run such IFFE scripts and create header files with properly defined configuration parameters.
4. Instrument C source code to include `FEATURE` header files and use `#define` symbols in these files to select code variants.
5. Restrain `FEATURE` file proliferation by limiting their use to libraries when possible.

By following the above steps during any port of a software system, porting knowledge is never forgotten. Indeed, such knowledge is coded in a form that is readily reusable in different software systems. In extreme cases `FEATURE` files generated on one platform may be used to bootstrap software on another. We have used this technique to port much of our software to Windows NT. The port started with `FEATURE` files from a mostly ANSI/POSIX system which were edited as necessary until `ksh` was up and running. Then, other software systems could be rebuilt with IFFE whose interpreter is written in the Bourne shell language [Bou78].

3. The IFFE language

An IFFE input file consists of a sequence of statements that define comments, options or probes. A comment statement starts with `#` and is ignored. An option statement is used to customize the execution behavior of the IFFE interpreter such as changing the compiler or resetting debugging level. The heart of the IFFE language is the probe statement. Below is its general form:

```
type name [ header ... ] [ library ... ] [ block ... ]
```

Here, *type* names the type of probe to apply, *name* names the object on which the probe is applied, *header* and *library* are optional comma-separated lists of headers and libraries to be passed to the compiler (non-existent ones are ignored), and *block* are optional multi-line blocks that define the probe's code.

type and *name* may be comma-separated lists in which case all *type*s are applied to all *name*s. Though *type* can be any value defined by users, probes for certain common types are provided by default. Below is a partial list of the common types.

`lib`: Checks if *name* is a function in the standard libraries.

`hdr`: Checks if `#include <`*name*`.h>` is valid.

`sys`: Checks if `#include <sys/`*name*`.h>` is valid.

`key`: Checks if *name* is a C language keyword.

`mac`: Checks if *name* is a C preprocessor macro.

`typ`: Checks if *name* is a type defined in `sys/types.h`, `stdlib.h` or `stddef.h`.

`cmd`: Checks if *name* is an executable in a standard directories such as `/bin` or `/etc`. If the command is found, the symbol `_cmd_`*name* is defined. In addition, each directory containing the command generates the symbol `_cmd_`*dir_name*.

The default output for a successful probe is shown below. Note that since the constructed output symbols must contain *type*, the names of the above default types are made short so that there is less chance of name conflicts in older compilers that restrict symbols to less than 8 characters.

```
#define _type_name 1   /* comment */
```

For example, the probe statement "`lib bcopy,memcpy`" checks to see if `bcopy()` and/or `memcpy()` are available in a standard library (most likely `/lib/libc.a`). On an old BSD Unix system, the output of this probe is likely to be:

```
#define _lib_bcopy 1   /* bcopy() in default lib(s) */
```

If the application code desires to use `memcpy()` exclusively then the probe output can be used to mimic or replace `memcpy ()` as follows:

```
#if _lib_bcopy && !_lib_memcpy
#define memcpy(to,from,size) (bcopy(from,to,size),to)
#endif
```

The optional *block*s in a probe statement are labeled and indicate actions to be done. Each block is of the form:

```
label{
    line
    ...
}end
```

Certain block labels indicate that the respective blocks contain actions to be done after a probe is executed. These labels are:

- `fail`: If the probe fails then the block is evaluated as a shell script and its output is copied to the output file.
- `pass`: If the probe succeeds then the default output is suppressed, the block is evaluated as a shell script and its output is copied to the output file.
- `note`: If the probe succeeds then the block is output as a single comment.
- `cat`: If the probe succeeds then the block is copied to the output file.

Other block labels mean that the respective blocks define probe code to override the respective default code templates if any. A probe is consider successful if it exits with status `0`. These block labels are:

- `run`: The block is run a shell script and the output is copied to the output file.
- `preprocess`: The block is preprocessed as a C program.
- `compile`: The block is compiled as a C program.
- `link`: The block is compiled and linked as a C program.
- `execute`: The block is compiled and linked as a C program and is then executed; the output is ignored.
- `output`: The block is compiled and linked as a C program, is then executed and the output is copied to the output file.

Below is an example of checking to see if the `mmap()` system call is available and if it does the job. In this case, the default probe code template to check for the existence of `mmap()` in some standard library (e.g., `/lib/libc.a`) is not good enough. The `execute` block indicates that the given program must be compiled and run to ensure that `mmap()` exists and works as expected. The probe success is defined by returning `0` at the end of execution.

```
lib mmap sys/types.h fcntl.h sys/mman.h execute{
    main(argc,argv)
    int   argc;
    char* argv[];
    {   int     fd;
        caddr_t p;
        if((fd = open(argv[0],0)) < 0)
            return 1;
        if(!(p = (caddr_t)mmap(0,1024,PROT_READ,MAP_SHARED,fd,0L)) ||
           p == ((caddr_t)-1) )
            return 1;
        return 0;
    }
}end
```

4. Writing and executing probes

A typical probe is a fragment of C code to be processed in some form. As discussed in Section 3, certain probe types come with default code templates but others must be supplied. This section discusses the style for writing C code in probe tests and briefly talks about the IFFE interpreter.

4.1. C code in IFFE probes

To eliminate duplication and ease the writing of probe code, IFFE automatically provides a number of preprocessor macros for code in the blocks (`preprocess`, `compile`, `link`, and `execute`). With proper use of these macros, code for probes can be written to be transparently compilable with different C language variants including K&R-C, ANSI-C, and C++ . The macros are:

`_STD_`: This symbol is `#define`d to `1` if the compiler is some flavor of ANSI-C or C++. Otherwise, it is defined to be `0`.

`_VOID_`: This is defined to be `void` for ANSI-C and C++ and `char` for older C.

`_ARG_((`*x*`))`: This macro function expands function prototypes depending on the underlying C language. Note that the extra pair of parentheses is required to avoid variable argument macro conflicts.

`_NIL_(`*type*`)`: This is a convenient macro that expands to `((`*type*`)0)`.

`_BEGIN_EXTERNS_`, `_END_EXTERNS_`:
These macros should be used around `extern` declarations to prevent their C names from being mangled by certain C++ implementations.

Below is another example probe taken from the *sfio* library. This probe checks to see if the routine `_cleanup()` of the *stdio* package is called when a program exits. *sfio* uses this information along with other information on exiting conventions of the local environment to configure its own clean-up procedure upon program exiting. Note that the type `exit` is application-defined.

```
exit cleanup note{ exit() calls _cleanup() }end execute{
    _BEGIN_EXTERNS_
    extern void exit _ARG_((int));
    extern void _exit _ARG_((int));
    _END_EXTERNS_
    void _cleanup() { _exit(0); }
    main() { exit(1); }
    }end
```

The probe works by explicitly calling from `main()` the `exit()` routine with an exit status `1` to signify probe failure. However, if `_cleanup()` is called implicitly, it will call `_exit()` which causes the program to exit with status `0` to signify probe success. Note that by necessity this probe must be both compiled and run. `_exit_cleanup` is defined on our local

SunOS 4.1 system. We shall not go into detail about why this probe is necessary. Suffice it to say that it was done when the *sfio* library was ported to an environment where there is no `atexit()`-like function and the code is expected to compile in a normal C environment but it may be linked with C++ code.

4.2. The IFFE interpreter

As IFFE is a part of the build procedure, it must be maximally portable. For this reason, the IFFE interpreter is written in the Bourne shell language which is supported on all known UNIX systems. It currently stands at about 1200 lines of code. The interpreter has a single option: if the first argument is "–" then the probe output is written to the standard output rather than the default `FEATURE/`*name*. Other arguments are interpreted as IFFE statements, where a ":" argument is the statement separator.

Below are a few typical interpreter invocations. The first one runs the probe tests in `features/lib`. The second one sets the compiler to `CC`, i.e., the C++ compiler, then runs the probe tests in `features/stdio.c`. The last one tests to see if `socket()` and `sys/socket.h` are available and writes the output to the terminal. This is a useful way to find out quickly certain information about the programming environment.

```
iffe run features/lib
iffe set cc CC : run features/stdio.c
iffe - lib,sys socket
```

5. Comparisons with other approaches

The idea of automatically configuring a software system by probing the native platform underlies the build procedure of many popular systems such as PERL [WS90] and older versions of KSH and NMAKE. Some have gone as far as writing code to make exhaustive lists of virtually all machine properties [Pem92]. The problem with this approach is that the lists often include much more than necessary and may cause unwanted side effects due to the large number of symbols. The programs that generate such lists are also susceptible to the usual problems in porting and maintenance. Another approach based on a combination of parameter and configuration files is reported in [TC92]. A cursory look at the examples given in the paper show that this approach relies on some scheme of broad platform classification (e.g., references to `bsd43` or `mips`). As observed in Section 2, this scheme does not always work and requires much more knowledge from the installer than necessary.

Closer in spirit to IFFE is the METACONFIG system by Larry Wall. This works by maintaining a glossary of symbols and a depository of probe units corresponding to the symbols in the glossary. Then, each application generates a shell script that includes all probes corresponding to source symbols that appear in the glossary. This script is packaged with the build procedure and run to generate the correct definitions of needed symbols each time the system is rebuilt. Though this is similar to the IFFE approach, there are fundamental differences between the systems at different usage levels as discussed below.

At the specification level, the symbols in the METACONFIG glossary are similar to those generated by IFFE from the *type* and *name* attributes of probe statements. However, because the glossary is centrally maintained, it can become arbitrarily large, cumbersome and difficult to master. By separating *type* and *name* as independent components of a symbol (Section 3) and defining a small number of default *type* probe code templates, IFFE reduces the effort to specify probes for such symbols. In fact, most IFFE probes for common objects (e.g., checking the existence of `bcopy()`) reduce to single lines of specification. Sharing of porting knowledge is done by reusing probe scripts. Thus, each application can define or acquire exactly the symbols and probes that it needs.

At the code shipment level, METACONFIG introduces an asymmetry between the provider and receiver of a software system because the receiver may lack the means to make the METACONFIG-generated script. If new probes are required during a port by the receiver, the only recourse is to modify this script which is not a simple procedure since the script can be quite large and complex. Further, since the script is automatically generated, it is also the wrong place to place such porting knowledge. IFFE scripts are treated as parts of the source code. The IFFE interpreter is usually shipped along with the source code. In this way, both the provider and receiver of a system see exactly the same thing. As the IFFE language is relatively simple, new probes required by porting can be easily recorded.

Finally, at the development level, the reliance on a single script for parametrization means that slight changes in probes may trigger long rebuilds since this script must be rebuilt and run. Though this is not a problem with software already advanced to a stable stage, it can be a serious nuisance during porting efforts. In addition, in an environment where parallel build is available (e.g., running `nmake` on a network of homogeneous machines), the METACONFIG-generated script can be the bottleneck and cause ineffective use of computing resource. The IFFE approach of making a correspondence between source probe files (in `features`) and generated headers (in `FEATURE`) holds an advantage because separate headers can be generated on a as-needed basis and then they can be simultanously generated on different processors if the environment allows. This approach also fits well with the general philosophy of build tools such as MAKE [Fel79] and NMAKE that certain objects are generated from other source objects.

6. Conclusion

As stated at the start of this paper, the portability problem boils down to finding out from a platform exactly which of its features are required and whether such features perform as expected. Any scheme of answering this question based on a broad classification of platforms (e.g., BSD vs. SYSV or SPARC vs. MIPS) is doomed to fail because modern environments tend to contain ad hoc mixtures of features. Even when a required feature is available, a bane to programmers is that its implementation quality can vary greatly from platform to platform. For example, the `mmap()` system call is a good alternative to `read()` for reading disk data on many modern UNIX systems because it avoids a buffer copy. But on certain platforms, `mmap()` simply does not work and on others, its performance can be worse than `read()`. This makes it hard to effect high quality implementation of critical software components such as the buffered I/O library *sfio*. IFFE and its accompanying programming style provide an effective solution by enabling programmers to target specific platform features and perform a variety of tests to determine their acceptability. The high level IFFE language also provides a convenient mechanism to record porting knowledge in a form that is easily shared among software developers.

7. References

[BK89] Morris Bolsky and David G. Korn. *The KornShell Command and Programming Language*. Prentice-Hall Inc., 1989.

[Bou78] S. R. Bourne. The Unix Shell. *AT&T Bell Laboratories Technical Journal*, 57(6):1971–1990, July 1978.

[Fel79] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):256–265, April 1979.

[Fow85] Glenn S. Fowler. The Fourth Generation Make. In *Proceedings of the USENIX 1985 Summer Conference*, pages 159–174, June 1985.

[FSV94] Glenn S. Fowler, John J. Snyder, and Kiem-Phong Vo. End-User Systems, Reusability, and High-Level Design. In *Proc. of the 1994 USENIX Symp. on Very High Level Languages*, October 1994.

[KV91] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer USENIX Conference*, pages 235–256. USENIX, 1991.

[Pem92] S. Pemberton. The Ergonomics of Software Porting. Technical Report CS-R9266, Center for Mathematics and Computer Science of the Mathematical Centre Foundation, Amsterdam, December 1992.

[TC92] D. Tilbrook and R. Crook. Large scale porting through parameterization. In *Proceedings of the USENIX 1992 Summer Conference*, 1992.

[Vo90] Kiem-Phong Vo. IFS: A Tool to Build Application Systems. *IEEE Software*, 7(4):29–36, July 1990.

[WS90] Larry Wall and Randal Schwartz. *Perl*. O'Reilly & Associates, 1990.

Biography

Glenn Fowler is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of NMAKE, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in Electrical Engineering, all from Virginia Tech, Blacksburg Virginia.

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer). Dave is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, KSH, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs Fellow award.

John J. Snyder is a Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories, where he has done UNIX systems administration and now works mostly on software for end-user systems. He received a Ph.D. in Econometrics from the University of Colorado in 1979. At that time he worked with FORTRAN on a Cray-1 and UNIX on a DEC PDP 11/70 at the National Center for Atmospheric Research in Boulder. After consulting in Mexico City for a couple of years, he joined AT&T in 1983.

Kiem-Phong Vo is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include aspects of graph theory and discrete algorithms and their applications in reusable and portable software tools. Aside from obscure theoretical works, Phong has worked on a number of popular software tools including the curses and malloc libraries in UNIX System V, sfio, a safe/fast buffered I/O library and DAG, a program to draw directed graphs. Phong joined Bell Labs in 1981 after receiving a Ph.D. in Mathematics from the University of California at San Diego. He received a Bell Labs Fellow award in 1991.