Portability of C Programs and the UNIX System*

S. C. JOHNSON
D. M. RITCHIE

ABSTRACT

Computer programs are portable to the extent that they can be moved to new computing environments with much less effort than it would take to rewrite them. In the limit, a program is perfectly portable if it can be moved at will with no change whatsoever. Recent C language extensions have made it easier to write portable programs. Some tools have also been developed that aid in the detection of nonportable constructions. With these tools many programs have been moved from the PDP-11 on which they were developed to other machines. In particular, the UNIX operating system and most of its software have been transported to the Interdata 8/32. The source-language representation of most of the code involved is identical in all environments.

I. INTRODUCTION

A program is portable to the extent that it can be easily moved to a new computing environment with much less effort than would be required to write it afresh. It may not be immediately obvious that lack of portability is, or needs to be, a problem. Of course, practically no assembly-language programs are portable. The fact is, however, that most programs, even in high-level languages, depend explicitly or implicitly on assumptions about such machine-dependent features as word and character sizes, character set, file system structure and organization, peripheral device handling, and many others. Moreover, few computer languages are understood by more than a handful of kinds of machines, and those that are (for example, Fortran and Cobol) tend to be rather limited in their scope, and, despite strong standards efforts, still differ considerably from one machine to another.

The economic advantages of portability are very great. In many segments of the computer industry, the dominant cost is development and maintenance of software. Any large organization, certainly including the Bell System, will have a variety of computers and will want to run the same program at many locations. If the program must be rewritten for each machine and maintained for each, software costs must increase. Moreover, the most effective hardware for a given job is not constant as time passes. If a nonportable program remains tied to obsolete hardware to avoid the expense of moving it, the costs are equally real even if less obvious. Finally, there can be considerable benefit in using machines from several manufacturers simply to avoid being utterly dependent on a single supplier.

Most large computer systems spend most of their time executing application programs; circuit design and analysis, network routing, simulation, data base applications, and text processing are particularly important at Bell Laboratories. For years, application programs have been written in high-level languages, but the programs that provide the basic software environment of computers (for example, operating systems, compilers, text editors, etc.) are still usually coded in assembly language. When the costs of hardware were large relative to the costs of software, there was perhaps some justification for this approach; perhaps an equally important reason was the lack of appropriate, adequately supported languages. Today hardware is relatively cheap, software is expensive, and any number of languages are capable of expressing well the algorithms required for basic system software. It is a mystery why the vast majority of computer manufacturers continue to generate so much assembly-language software.

The benefits of writing software in a well-designed language far exceed the costs. Aside from potential portability, these benefits include much smaller development and maintenance costs. It is true that a

* Originally published in The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp 2021-2048.

penalty must be paid for using a high-level language, particularly in memory space occupied. The cost in time can usually be controlled: experience shows that the time-critical part of most programs is only a few percent of the total code. Careful design allows this part to be efficient, while the remainder of the program is unimportant.

Thus, we take the position that essentially all programs should be written in a language well above the level of machine instructions. While many of the arguments for this position are independent of portability, portability is itself a very important goal; we will try to show how it can be achieved almost as a by-product of the use of a suitable language.

We have recently moved the UNIX system kernel, together with much of its software, from its original host machine (DEC PDP-11) to a very different machine (Interdata 8/32). Almost all the programs involved are written in the C language [1, 2] and almost all are identical on the two systems. This paper discusses some of the problems encountered, and how they were solved by changing the language itself and by developing tools to detect and resolve nonportable constructions. The major lessons we have learned, and that we hope to teach, are that portable programs are good programs for more reasons than that they are portable, and that making programs portable costs some intellectual effort but need not degrade their performance.

II. HISTORY

The Computing Science Research Center at Bell Laboratories has been interested in the problems and technologies of program portability for over a decade. Altran [3] is a substantial (25,000 lines) computer algebra system, written in Fortran, which was developed with portability as one of its primary goals. Altran has been moved to many incompatible computer systems; the effort involved for each move is quite moderate. Out of the Altran effort grew a tool, the PFORT verifier [4], that checks Fortran programs for adherence to a strict set of programming conventions. Most importantly, it detects (where possible) whether the program conforms to the ANSI standard for Fortran [5], but because many compilers fail to accept even standard-conforming programs, it also remarks upon several constructions that are legal but nevertheless nonportable. Successful passage of a program through PFORT is an important step in assuring that it is portable. More recently, members of the Computer Science Research Center and the Computing Technology Center jointly created the PORT library of mathematical software [6]. Implementation of PORT required research not merely into the language issues, but also into deeper questions of the model of floating point computations on the various target machines. In parallel with this work, the development at Bell Laboratories of Snobol4 [7] marks one of the first attempts at making a significant compiler portable. Snobol4 was successfully moved to a large number of machines, and, while the implementation was sometimes inefficient, the techniques made the language widely available and stimulated additional work leading to more efficient implementations.

III. PORTABILITY OF C PROGRAMS - INITIAL EXPERIENCES

C was developed for the PDP-11 on the UNIX system in 1972. Portability was not an explicit goal in its design, even though limitations in the underlying machine model assumed by the predecessors of C made us well aware that not all machines were the same [2]. Less than a year later, C was also running on the Honeywell 6000 system at Murray Hill. Shortly thereafter, it was made available on the IBM 310 series machines as well. The compiler for the Honeywell was a new product[8]. but the IBM compiler was adapted from the PDP-11 version, as were compilers for several other machines.

As soon as C compilers were available on other machines, a number of programs, some of them quite substantial, were moved from UNIX to the new environments. In general, we were quite pleased with the ease with which programs could be transferred between machines. Still, a number of problem areas were evident. To begin with, the C language was growing and developing as experience suggested new and desirable features. It proved to be quite painful to keep the various C compilers compatible, the Honeywell version was entirely distinct from the PDP-11 version, and the IBM version had been adapted, with many changes, from a by-then obsolete version of the PDP-11 compiler. Most seriously, the operating system interface caused far more trouble for portability than the actual hardware or language differences themselves. Many of the UNIX primitives were impossible to imitate on other operating systems; moreover, some conventions on these other operating systems (for example, strange file formats and record-oriented

I/O) were difficult to deal with while retaining compatibility with UNIX. Conversely, the I/O library commonly used sometimes made UNIX conventions excessively visible--for example, the number 518 often found its way into user programs as the size, in bytes, of a particularly efficient I/O buffer structure.

Additional problems in the compilers arose from the decision to use the local assemblers, loaders, and library editors on the host operating systems. Surprisingly often, they were unable to handle the code most naturally produced by the C compilers. For example, the semantics of possibly initialized external variables in C was quite consciously designed to be implementable in a way identical to Fortran's COMMON blocks to guarantee its portability. It was an unpleasant surprise to discover that the Honeywell assembler would allow at most 61 such blocks (and hence external variables) and that the IBM link-editor preferred to start external variables on even 4096-byte boundaries. Software limitations in the target systems complicated the compilers and, in one case, the problems with external variables just mentioned, forced changes in the C language itself.

IV. THE UNIX PORTABILITY PROJECT

The realization that the operating systems of the target machines were as great an obstacle to portability as their hardware architecture led us to a seemingly radical suggestion: to evade that part of the problem altogether by moving the operating system itself. Transportation of an operating system and its software between non-trivially different machines is rare, but not unprecedented [9-13]. Our own situation was a bit different in that we already had a moderately large, complete, and mature system in wide use at many installations. We could not (or at any rate did not want to) start afresh and redesign the language, the operating system interfaces, and the software. It seemed, though, that despite some problems in each we had a good base to build on.

Our project had three major goals:

(i) To write a compiler for C that could be changed without grave difficulty to generate code for a variety of machines.

(ii) To refine and extend the C language to make most C programs portable to a wide variety of machines, mechanically identifying non-portable constructions where possible.

(iii) To revise or recode a substantial portion of UNIX in portable C, detecting and isolating machine dependencies, and demonstrate its portability by moving it to another machine.

By pursuing each goal, we hoped to attain a corresponding benefit:

(i) A C compiler adaptable to other machines (independently of UNIX), that puts into practice some recent developments in the theory of code generation.

(ii) Improved understanding of the proper design of languages that, like C, operate on a level close to that of real machines but that can be made largely machine-independent.

(iii) A relatively complete and usable implementation of UNIX on at least one other machine, with the hope that subsequent implementations would be fairly straightforward.

We selected the Interdata 8/32 computer to serve as the initial target for the system portability research. It is a 32-bit computer whose design resembles that of the IBM System/360 and /370 series machines, although its addressing structure is rather different; in particular, it is possible to address any byte in virtual memory without use of a base register. For the portability research, of course, its major feature is that it is *not* a PDP-11. In the longer term, we expect to find it especially useful for solving problems, often drawn from numerical analysis, that cannot be handled on the PDP-11 because of its limited address space.

Two portability projects besides those referred to above are particularly interesting. In the period 1976-1977, T. L. Lyon and his associates at Princeton adapted the UNIX kernel to run in a virtual machine partition under VM/370 on an IBM System/370 [14]. Enough software was also moved to demonstrate the feasibility of the effort, though no attempt was made to produce a complete, working system. In the midst of our own work on the Interdata 8/32, we learned that a UNIX portability project, for the similar Interdata 7/32, was under way at the University of Wollongong in Australia [15]. Since everything we know of this effort was discovered in discussion with its major participant, Richard Miller [16], we will remark only that the transportation route chosen was markedly different from ours. In particular, an Interdata C compiler was adapted from the PDP-11 compiler, and was moved as soon as possible to the Interdata, where it ran under the manufacturer's operating system. Then the UNIX kernel was moved in pieces, first running with dummy device drivers as a task under the Interdata system, and only at the later stages independently. This approach, the success of which must be scored as a real *tour de force*, was made necessary by the 100 kilometers separating the PDP-11 in Sydney from the Interdata in Wollongong.

4.1 Project chronology

Work began in the early months of 1977 on the compiler, assembler, and loader for the Interdata machine. Soon after its delivery at the end of April 1977, we were ready to check out the compiler. At about the same time, the operating system was being scrutinized for nonportable constructions. During May, the Interdata-specific code in the kernel was written, and by June, it was working well enough to begin moving large amounts of software; T. L. Lyon aided us greatly by tackling the bulk of this work. By August, the system was unmistakably UNIX, and it was clear that, as a research project, the portability effort had succeeded, though there were still programs to be moved and bugs to be stamped out. From late summer until October 1977, work proceeded more slowly, owing to a combination of hardware difficulties and other claims on our time; by the spring of 1978 the portability work as such was complete. The remainder of this paper discusses how success was achieved.

V. SOME NON-GOALS

It was and is clear that the portability achievable cannot approach that of Altran, for example, which can be brought up with a fortnight of effort by someone skilled in local conditions but ignorant of Altran itself. In principle, all one needs to implement Altran is a computer with a standard Fortran compiler and a copy of the Altran system tape; to get it running involves only defining of some constants characterizing the machine and writing a few primitive operations in assembly language.

In view of the intrinsic difficulties of our own project, we did not feel constrained to insist that the system be so easily portable. For example, the C compiler is not bootstrapped by means of a simple interpreter for an intermediate language; instead, an acceptably efficient code generator must be written. The compiler is indeed designed carefully so as to make changes easy, but for each new machine it inevitably demands considerable skill even to decide on data representations and run-time conventions, let alone the code sequences to be produced. Likewise, in the operating system, there are many difficult and inevitably machine-dependent issues, including especially the treatment of interrupts and faults, memory management, and device handling. Thus, although we took some care to isolate the machine-dependent portions of the operating system into a set of primitive routines, implementation of these primitives involves deep knowledge of the most recondite aspects of the target machine.

Moreover, we could not attempt to make the portable UNIX system compatible with software, file formats, or inadequate character sets already existing on the machine to which it is moved; to promise to do so would impossibly complicate the project and, in fact, might destroy the usefulness of the result. If UNIX is to be installed on a machine, its way of doing business must be accepted as the right way; afterwards, perhaps, other software can be made to work.

VI. THE PORTABLE C COMPILER

The original C compiler for the PDP-11 was not designed to be easy to adapt for other machines. Although successful compilers for the IBM System/370 and other machines were based on it, much of the modification effort in each case, particularly in the early stages, was concerned with ridding it of assumptions about the PDP-11. Even before the idea of moving UNIX occurred to us, it was clear that C was successful enough to warrant production of compilers for an increasing variety of machines. Therefore, one of the authors (SCJ) undertook to produce a new compiler intended from the start to be easily modified. This new compiler is now in use on the IBM System/370 under both OS and TSS, the Honeywell 6000, the Interdata 8/32, the SEL86 the Data General Nova and Eclipse, the DEC VAX-11/780, and a Bell System processor. Versions are in progress for the Intel 8086 microprocessor and other machines.

The degree of portability achieved by this compiler is satisfying. In the Interdata 8/32 version, there are roughly 8,000 lines of source code. The first pass, which does syntax and lexical analysis and symbol table management, builds expression trees, and generates a bit of machine-dependent code such as subroutine prologues and epilogues, consists of 4,600 lines of code, of which 600 are machine-dependent. In the second pass, which does the bulk of the code generation, 1,000 out of 3,400 lines are machine-dependent. Thus, out of a total of 8,000 lines, 1,600, or 20 percent, are machine-dependent; the remaining 80 percent are shared with the Honeywell, IBM, and other compilers. As the Interdata compiler becomes more carefully tuned, the machine-dependent figures will rise somewhat; for the IBM, the machine-dependent fraction is 22 percent; for the Honeywell, 25 percent.

These figures both overstate and understate the true difficulty of moving the compiler. They represent the size of those source files that contain machine-dependent code; only a half or a third of the lines in many machine-dependent functions actually differ from machine to machine, because most of the routines involved remain similar in structure. As an example, routines to output branches, align location counters, and produce function prologues and epilogues have a clear machine-dependent component, but nevertheless are logically very similar for all the compilers. On the other hand, as we discuss below, the hardest part of moving the compiler is not reflected in the number of lines changed, but is instead concerned with understanding the code generation issues, the C language, and the target machine well enough to make the modifications effectively.

The new compiler is not only easily adapted to a new machine, it has other virtues as well. Chief among these is that all versions share so much code that maintenance of all versions simultaneously involves much less work than would maintaining each individually. For example, if a bug is discovered in the machine-independent portion, the repair can be made to all versions almost mechanically. Even if the language itself is changed, it is often the case that most of the job of installing the change is machine-independent and usable for all versions. This has allowed the compilers for all machines to remain compatible with a minimum of effort.

The interface between the two passes of the portable C compiler consists of an intermediate file containing mostly representations of expression trees together with character representations of stereotyped code for subroutine prologues and epilogues. Thus a different first pass can be substituted provided it conforms to the interface specifications. This possibility allowed S. I. Feldman to write a first pass that accepts the Fortran 77 language instead of C. At the moment, the Fortran front-end has two versions (which differ by about as much as do the corresponding first passes for C) that feed the code generators for the PDP-11 and the Interdata machines. Thus we apparently have not only the first, but the first two implementations of Fortran 77.

6.1 Design of the portable compiler

Most machine-dependent portions of a C compiler fall into three categories.

(i) Storage allocation.

(ii) Rather stereotyped code sequences for subroutine entry points and exits, switches, labels, and the like.

(iii) Code generation for expressions.

For the most part, storage allocation issues are easily parameterized in terms of the number of bits required for objects of the various types and their alignment requirements. Some issues, like addressability on the IBM 360 and 310 series, cause annoyance, but generally there are few problems in this area. The calling sequence is very important to the efficiency of the result and takes considerable knowledge and imagination to design properly. However, once designed, the calling sequence code and the related issue of stack frame layout are easy to cope with in the compiler.

Generating optimal code for arithmetic expressions, even on idealized machines, can be shown theoretically to be a nearly intractable problem. For the machines we are given in real life, the problem is even harder. Thus, all compilers have to compromise a bit with optimality and engage in heuristic algorithms to some extent, in order to get acceptably efficient code generated in a reasonable amount of time.

The design of the code generator was influenced by a number of goals, which in turn were influenced by recent theoretical work in code generation. It was recognized that there was a premium in being able to get the compiler up and working quickly; it was also felt, however, that this was in many ways less important than being able to evolve and tune the compiler into a high-quality product as time went on.

Particularly with operating system code, a "quick and dirty" implementation is simply unacceptable. It was also recognized that the compiler was likely to be applied to machines not well understood by the compiler writer that might have inadequate or nonexistent debugging facilities. Therefore, one goal of the compiler was to permit it to be largely self-checking. Rather than produce incorrect code, we felt it far preferable for the compiler to detect its own inadequacies and reject the input program.

This goal was largely met. The compiler for the Interdata 8/32 was working within a couple of weeks after the machine arrived; subsequently, several months went by with very little time lost due to compiler bugs. The bug level has remained low, even as the compiler has begun to be more carefully tuned; many of the bugs have resulted from human error (e.g., misreading the machine manual) rather than actual compiler failure.

Several techniques contribute considerably to the general reliability of the compiler. First, a conscious attempt was made to separate information about the machine (e.g., facts such as "there is an add instruction that adds a constant to a register and sets the condition code") from the strategy, often heuristic, that makes use of these facts (e.g., if an addition is to be done, first compute the left-hand operand into a register). Thus, as the compiler evolves, more effort can be put into improving the heuristics and the recognition of important special cases, while the underlying knowledge about the machine operations need not be altered. This approach also improves portability, since the heuristic programs often remain largely unchanged among similar machines, while only the detailed knowledge about the format of the instructions (encoded in a table) changes.

During compilation of expressions, a model of the state of the compilation process, including the tree representing the expression being compiled and the status of the machine registers, is maintained by the compiler. As instructions are emitted, the expression tree is simplified. For example, the expression **a = b+c** might first be transformed into **a =** *register* **+b** as a load instruction for **a** is generated, then into **a =** *register* when an add is produced. The possible transformations constitute the "facts" about the machine: the order in which they are applied correspond to the heuristics. When the input expression has been completely transformed into nothing, the expression is compiled. Thus, a good portion of the initial design of a new version of the compiler is concerned with making the model within the compiler agree with the actual machine by building a table of machine operations and their effects on the model. When this is done correctly, one has a great deal of confidence that the compiler will produce correct code, if it produces any at all.

Another useful technique is to partition the code generation job into pieces that interact only through well-defined paths. One module worries about breaking up large expressions into manageable pieces, and allocating temporary storage locations when needed. Another module worries about register allocation. Finally, a third module takes each "manageable" piece and the register allocation information, and generates the code. The division between these pieces is strict; if the third module discovers that an expression is "unmanageable," or a needed register is busy, it rejects the compilation. The division enforces a discipline on the compiler which, while not really restricting its power, allows for fairly rapid debugging of the compiler output.

The most serious drawback of the entire approach is the difficulty of proving any form of "completeness" property for the compiler--of demonstrating that the compiler will in fact successfully generate code for all legal C programs. Thus, for example, a needed transformation might simply be missing, so that there might be no way to further simplify some expression. Alternatively, some sequence of transformations might result in a loop, so that the same expression keeps reappearing in a chain of transformations. The compiler detects these situations by realizing that too many passes are being made over the expression tree, and the input is rejected. Unfortunately, detection of these possibilities is difficult to do in advance because of the use of heuristics in the compiler algorithms. Currently, the best way of ensuring that the compiler is acceptably complete is by extensive testing.

6.2 Testing the compiler

We ordered the Interdata 8/32 without any software at all, so we first created a very crude environment that allowed stand-alone programs to be run; all interrupts, memory mapping, etc., were turned off. The compiler, assembler, and loader ran on the PDP-11, and the resulting executable files were transferred to the Interdata for testing. Primitive routines permitted individual characters to be written on the console.

In this environment, the basic stack management of the compiler was debugged, in some cases by single-stepping the machine. This was a painful but short period.

After the function call mechanism was working, other short tests established the basic sanity of simple conditionals, assignments, and computations. At this point, the stand-alone environment could be enriched to permit input from the console and more informative output such as numbers and character strings, so ordinary C programs could be run. We solicited such programs, but found few that did not depend on the file system or other operating system features. Some of the most useful programs at this stage were simple games that pitted the computer against a human; they frequently did a large amount of computing, often with quite complicated logic, and yet restricted themselves to simple input and output. A number of compiler bugs were found and fixed by running games. After these tests, the compiler ceased to be an explicit object of testing, and became instead a tool by which we could move and test the operating system.

Some of the most subtle problems with compiler testing come in the maintenance phase of the compiler, when it has been tested, declared to work, and installed. At this stage, there may be some interest in improving the code quality as well as fixing the occasional bug. An important tool here is regression testing; a collection of test programs are saved, together with the previous compiler output.

Before a new compiler is installed, the new compiler is fed these test programs, the new output is compared with the saved output, and differences are noted. If no differences are seen, and a compiler bug has been fixed or improvement made, the testing process is incomplete, and one or more test programs are added. If differences are detected, they are carefully examined. The basic problem is that frequently, in attempting to fix a bug, the most obvious repair can give rise to other bugs, frequently breaking code that used to work. These other bugs can go undetected for some time, and are very painful both to the users and the compiler writer. Thus, regression tests attempt to guard against introducing new bugs while fixing old ones.

The portable compiler is sufficiently self-checked that many potential compiler bugs were detected before the compiler was installed by the simple expedient of turning the compiler loose on a large amount (tens of thousands of lines) of C source code. Many constructions turned up there that were undreamed of by the compiler writer, and often mishandled by the compiler.

It is worth mentioning that this kind of testing is easily carried out by means of the standard commands and features in the UNIX system. In particular, C source programs are easily identified by their names, and the UNIX shell provides features for applying command sequences automatically to each of a list of files in turn. Moreover, powerful utilities exist to compare two similar text files and produce a minimal list of differences. Finally, the compiler produces assembly code that is an ordinary text file readable by all of the usual utilities. Taken together, these features make it very simple to invent test drivers. For example, it takes only a half-dozen lines of input to request a list of differences between the outputs of two versions of the compiler applied to tens (or hundreds) of source files. Perhaps even more important, there is little or no output when the compilers compare exactly. On many systems, the "job control language" required to do this would be so unpleasant as to insure that it would not be done. Even if it were, the resulting hundreds of pages of output could make it very difficult to see the places where the compiler needed attention. The design of the portable C compiler is discussed more thoroughly in Ref. 17.

VII. LANGUAGE AND COMPILER ISSUES

We were favorably impressed, even in the early stages, by the general ease with which C programs could be moved to other machines. Some problems we did encounter were related to weaknesses in the C language itself, so we undertook to make a few extensions.

C had no way of accounting in a machine-independent way for the overlaying of data. Most frequently, this need comes up in large tables that contain some parts having variable structure. As an invented example, a compiler's table of constants appearing in a source program might have a flag indicating the type of each constant followed by the constant's value, which is either integer or floating. The C language as it existed allowed sufficient cheating to express the fact that the possible integer and floating value might be overlaid (both would not exist at once), but it could not be expressed portably because of the inability to express the relative sizes of integers and floating-point data in a machine-independent way. Therefore, the union declaration was added; it permits such a construction to be expressed in a natural and portable manner. Declaring a union of an integer and a floating point number reserves enough storage to hold either, and forces such alignment properties as may be required to make this storage useful as both an integer and a floating point number. This storage may be explicitly used as either integer or floating point by accessing it with the appropriate descriptor tag.

Another addition was the **typedef** facility, which in effect allows the types of objects to be easily parameterized. **typedef** is used quite heavily in the operating system kernel, where the types of a number of different kinds of objects, for example, disk addresses, file offsets, device numbers, and times of day, are specified only once in a header file and assigned to a specific name; this name is then used throughout. Unlike some languages, C does not permit definition of new operations on these new types; the intent was increased parameterization rather than true extensibility. Although the C language did benefit from these extensions, the portability of the average C program is improved more by restricting the language than by extending it. Because it descended from typeless languages, C has traditionally been rather permissive in allowing dubious mixtures of various types; the most flagrant violations of good practice involved the confusion of pointers and integers. Some programs explicitly used character pointers to simulate unsigned integers; on the PDP-11 the two have the same arithmetic properties. Type unsigned was introduced into the language to eliminate the need for this subterfuge.

More often, type errors occurred unconsciously. For example, a function whose only use of an argument is to pass it to a subfunction might allow the argument to be taken to be an integer by default. If the top-level actual argument is a pointer, the usage is harmless on many machines, but not type-correct and not, in general, portable.

Violations of strict typing rules existed in many, perhaps most, of the programs making up the entire stock of UNIX system software. Yet these programs, representing many tens of thousands of lines of source code, all worked correctly on the PDP-11 and in fact would work on many other machines, because the assumptions they made were generally, though not universally, satisfied. It was not feasible simply to declare all the suspect constructions illegal. Instead, a separate program was written to detect as many dubious coding practices as possible. This program, called **lint**, picks bits of fluff from programs in much the same way as the PFORT verifier mentioned above. C programs acceptable to lint are guaranteed to be free from most common type errors; lint also checks syntax and detects some logical errors, such as uninitialized variables, unused variables, and unreachable code.

There are definite advantages in separating program-checking from compilation. First, lint was easy to produce, because it is based on the portable compiler and thus shares the machine independent code of the first pass with the other versions of the compiler. More important, the compilers, large programs anyway, are not burdened with a great deal of checking code which does not necessarily apply to the machine for which they are running. A good example of extra capability feasible in lint but probably not in the compilers themselves is checking for inter-program consistency. The C compilers all permit separate compilation of programs in several files, followed by linking together of the results. lint (uniquely) checks consistency of declarations of external variables, functions, and function arguments among a set of files and libraries.

Finally, lint itself is a portable program, identical on all machines. Although care was taken to make it easy to propagate changes in the machine-independent parts of the compilers with a minimum of fuss, it has proved useful for the sometimes complicated logic of lint to be totally decoupled from the compilers. lint cannot possibly affect their ability to produce code; if a bug in lint turns up, its output can be ignored and work can continue simply by ignoring the spurious complaints. This kind of separation of function is characteristic of UNIX programs in general. The compiler's one important job is to generate code; it is left to other programs to print listings, generate cross-reference tables, and enforce style rules.

VIII. THE PORTABILITY OF THE UNIX KERNEL

The UNIX operating system kernel, or briefly the operating system, is the permanently resident program that provides the basic software environment for all other programs running on the machine. It implements the "system calls" by which user's programs interact with the file system and request other services, and arranges for several programs to share the machine without interference. The structure of the UNIX operating system kernel is discussed elsewhere in this issue [18, 19]. To many people, an operating system may seem the very model of a nonportable program, but in fact a major portion of UNIX and other well-written operating systems consists of machine-independent algorithms: how to create, read, write, and delete files, how to decide who to run and who to swap, and so forth. If the operating system is viewed as a large C program, then it is reasonable to hope to apply the same techniques and tools to it that we apply to move more modest programs. The UNIX kernel can be roughly divided into three sections according to their degree of portability.

8.1 Assembly-language primitives

At the lowest level, and least portable, is a set of basic hardware interface routines. These are written in assembly language, and consist of about 800 lines of code on the Interdata 8/32. Some of them are callable directly from the rest of the system, and provide services such as enabling and disabling interrupts, invoking the basic I/O operations, changing the memory map so as to switch execution from one process to another, and transmitting information between a user process's address space and that of the system. Most of them are machine-independent in specification, although not implementation. Other assembly-language routines are not called explicitly but instead intercept interrupts, traps, and system calls and turn them into C-style calls on the routines in the rest of the operating system.

Each time UNIX is moved to a new machine, the assembly language portion of the system must be rewritten. Not only is the assembly code itself machine-specific, but the particular features provided for memory mapping, protection, and interrupt handling and masking differ greatly from machine to machine. In moving from the PDP-11 to the Interdata 8/32, a huge preponderance of the bugs occurred in this section. One reason for this is certainly the usual sorts of difficulties found in assembly-language programming: we wrote loops that did not loop or looped forever, garbled critical constants, and wrote plausible-looking but utterly incorrect address constructions. Lack of familiarity with the machine led us to incorrect assumptions about how the hardware worked, and to inefficient use of available status information when things went wrong.

Finally, the most basic routines for multi-programming, those that pass control from one process to another, turned out (after causing months of nagging problems) to be incorrectly specified and actually unimplementable correctly on the Interdata, because they depended improperly on details of the register-saving mechanism of the calling sequence generated by the compiler. These primitives had to be redesigned; they are of special interest not only because of the problems they caused, but because they represent the only part of the system that had to be significantly changed, as distinct from expressed properly, to achieve portability.

8.2 Device drivers

The second section of the kernel consists of device drivers, the programs that provide the interrupt handling, I/O command processing, and error recovery for the various peripheral devices connected to the machine. On the Interdata 8/32 the total size of drivers for the disk, magnetic tape, console typewriter, and remote typewriters is about 1100 lines of code, all in C. These programs are, of course, machine-dependent, since the devices are.

The drivers caused far fewer problems than did the assembly language programs. Of course, they already had working models on the PDP-11, and we had faced the need to write new drivers several times in the past (there are half a dozen disk drivers for various kinds of hardware attached to the PDP-11). In adapting to the Interdata, the interface to the rest of the system survived unchanged, and the drivers themselves shared their general structure, and even much code, with their PDP-11 counterparts. The problems that occurred seem more related to the general difficulty of dealing with the particular devices than in expressing what had to be done.

8.3 The remainder of the system

The third and remaining section of the kernel is the largest. It is all written in C, and for the Interdata 8/32 contains about 7,000 lines of code. This is the operating system proper, and clearly represents the bulk of the code. We hoped that it would be largely portable, and as it turned out our hopes were justified. A certain amount of work had to be done to achieve portability. Most of it was concerned with making sure that everything was declared properly, so as to satisfy lint, and with replacing constants by parameters. For example, macros were written to perform various unit conversions previously written out explicitly: byte counts to memory segmentation units and to disk blocks, etc. The important data types used within the system were identified and specified using typedef: disk offsets, absolute times, internal device names, and the like. This effort was carried out by K. Thompson.

Of the 7,000 lines in this portion of the operating system, only about 350 are different in the Interdata and PDP-11 versions; that is, they are 95 percent identical. Most of the differences are traceable to one of three areas.

(i) On the PDP-11, the subroutine call stack grows towards smaller addresses, while on the Interdata it grows upwards. This leads to different code when increasing the size of a user stack, and especially when creating the argument list for an inter-program transfer (exec system call) because the arguments are placed on the stack.

(ii) The details of the memory management hardware on the two machines are different, although they share the same general scheme.

(iii) The routine that handles processor traps (memory faults, etc.) and system calls is rather different in detail on the two machines because the set of faults is not identical, and because the method of argument transmission in system calls differs as well.

We are extremely gratified by the ease with which this portion of the system was transferred. Only a few problems showed up in the code that was not changed; most were in the new code written specifically for the Interdata. In other words, what we thought would be portable did in fact move without trouble.

Not everything went perfectly smoothly, of course. Our first set of major problems involved the mechanics of transferring test systems and other programs from the PDP-11 to the Interdata 8/32 and debugging the result. Better communications between the machines would have helped considerably. For a period, installing a new Interdata system meant creating an 800 BPI tape on the sixth-floor PDP-11, carrying the tape to another PDP-11 on the first floor to generate a 1600 BPI version, and finally lugging the result to the fifth-floor Interdata. For debugging, we would have been much aided by a hardware interface between the PDP-11 and the front panel of the Interdata to allow remote rebooting. This class of problems is basically our own fault, in that we traded the momentary ease of not having to write communications software or build hardware for the continuing annoyance of carrying tapes and hands-on debugging.

Another class of problems seems impossible to avoid, since it stems from the basic differences in the representation of information on the two machines. In the machines at issue, only one difference is important: the PDP-11 addresses the two bytes in a 16 the Interdata the first byte in a 16-bit half-word is the most significant 8 bits. Since all the interfaces between the two machines are byte-serial, the effect is best described by saying that when a true character stream is transmitted between them, all is well; but if integers are sent, the bytes in each half-word must be swapped. Notice that this problem does not involve portability in the sense in which it has been used throughout this paper; very few C programs are sensitive to the order in which bytes are stored on the machine on which they are running. Instead it complicates "portability" in its root meaning wherein files are carried from one machine to the other. Thus, for example, during the initial creation of the Interdata system we were obliged to create, on the PDP-11, an image of a file system disk volume that would be copied to tape and thence to the Interdata disk, where it would serve as an actual file system for the latter machine. It required a certain amount of cleverness to declare the data structures appropriately and to decide which bytes to swap.

The ordering of bytes in a word on the PDP-11 is somewhat unusual, but the problem it poses is quite representative of the difficulties of transferring encoded information from machine to machine. Another example is the difference in representation of floating-point numbers between the PDP-11 and the Interdata. The assembler for the Interdata, when it runs on the PDP-11, must invoke a routine to convert the "natural" PDP-11 notation to the foreign notation, but of course this conversion must not be done when the assembler is run on the Interdata itself. This makes the assembler necessarily non-portable, in the sense that it must execute different code sequences on the two machines. However, it can have a single source representation by taking advantage of conditional compilation depending on where it will run.

This kind of problem can get much worse: how are we to move UNIX to a target machine with a 36-bit word length, whose machine word cannot even be represented by long integers on the PDP-11 ? Nevertheless, it is worth emphasizing that the problem is really vicious only during the initial bootstrapping phase; all the software should run properly if only it can be moved once!

IX. TRANSPORTATION OF THE SOFTWARE

Most UNIX code is in neither the operating system itself nor the compiler, but in the many user-level utilities implementing various commands and in subroutine libraries. The sheer bulk of the programs involved (about 50,000 lines of source) meant that the amount of work in transportation might be considerable, but our early experience, together with the small average size of each individual program, convinced us that it would be manageable. This proved to be the case.

Even before the advent of the Interdata machine, it was realized, as mentioned above, that many programs depended to an undesirable degree not only on UNIX I/O conventions but on details of particularly favorable buffering strategies for the PDP-11. A package of routines, called the "portable I/O library," was written by M. E. Lesk [20] and implemented on the Honeywell and IBM machines as well as the PDP-11 in a generally successful effort to overcome the deficiencies of earlier packages. This library too proved to have some difficulties, not in portability, but in time efficiency and space required. Therefore a new package of routines, dubbed the "standard I/O library," was prepared. Similar in spirit to the portable library, it is somewhat smaller and much faster. Thus, part of the effort in moving programs to the Interdata machine was devoted to making programs use the new standard I/O library. In the simplest cases, the effort involved was nil, since the fundamental character I/O functions have the same names in all libraries.

Next, each program had to be examined for visible lack of portability. Of course, lint was a valuable tool here. Programs were also scrutinized by eye to detect dubious constructions. Often these involved constants. For example, on the 16-bit PDP-11 the expression

x & 0177770

masks off all but the last three bits of x, since 0177770 is an octal constant. This is almost certainly better expressed

x & ˜07

(where ˜ is the ones-complement operator) because the latter expression actually does yield the last three bits of x independently of the word length of the machine. Better yet, the constant should be a parameter with a meaningful name.

UNIX software has a number of conventional data structures, ranging from objects returned or accepted by the operating system kernel (such as status information for a named file) to the structure of the header of an executable file. Programs often had a private copy of the declaration for each such structure they used, and often the declaration was nonportable. For example, an encoded file mode might be declared **int** on the 16-bit PDP-11, but on the 32-bit Interdata machine, it should be specified as **short**, which is unambiguously 16 bits. Therefore, another major task in making the software portable was to collect declarations of all structures common to several routines, to put the declarations in a standard place, and to use the **include** facility of the C preprocessor to insert them in the source program. The compiler for the PDP-11 and the cross-compiler for the Interdata 8/32 were adjusted to search a different standard directory to find the canned declarations appropriate to each.

Finally, an effort was made to seek out frequently occurring patches of code and replace them by standard subroutines, or create new subroutines where appropriate. It turned out, for example, that several programs had built-in subroutines to find the printable user name corresponding to a numerical user ID. Although in each case the subroutine as written was acceptably portable to other machines, the function it performed was not portable in time across changes in the format of the file describing the name-number correspondence; encapsulating the translation function insulated the program against possible changes in a data base.

X. THE MACHINE MODEL FOR C

One of the hardest parts of designing a language in which to write portable programs is deciding which properties are guaranteed to remain invariant. Likewise, in trying to develop a portable operating system, it is very hard to decide just what properties of the underlying machine can be depended on. The design questions in each case are many in number; moreover, the answer to each individual question may involve tradeoffs that are difficult to evaluate in advance. Here we try to show the nature of these tradeoffs and what sort of compromises are required.

Designing a language in which every program is portable is actually quite simple: specify precisely the meaning of every legal program, as well as what programs are legal. Then the portability problem does not exist: by definition, if a correct program fails on some machine, the language has not been implemented properly. Unfortunately, a language like C that is intended to be used for system programming is not very adaptable to such a Procrustean approach, mainly because reasonable efficiency is required. Any well-defined language can be implemented precisely on any general-purpose computer, but the implementation may not be usable in practice if it implies use of an interpreter rather than machine instructions. Thus, with both language and operating system design, one must strike a balance between convenient and powerful features and the ease of implementing them efficiently on a variety of machines. At any point, some machine may be found on which some feature is very expensive to provide, and a decision must be made whether to modify the feature, and thus compromise the portability of programs that use it, or to insist that the meaning is immutable and must be preserved. In the latter case portability is also compromised since the cost of using the feature may be so high that no one can afford the programs that use it, or the people attempting to implement the feature on the new machine give up in despair.

Thus a language definition implies a model of the machine on which programs in the language will run. If a real machine conforms well to the model, then an implementation on that machine is likely to be efficient and easily written; if not, the implementation will be painful to provide and costly to use. Here we shall consider the major features of the abstract C machine that have turned out to be most relevant so far.

10.1 Integers

Probably the most frequent operations are on integers consisting of various numbers of bits. Variables declared **short** are at least 16 bits in length; those declared **long** are at least 32 bits. Most are declared **int**, and must be at least as precise as short integers, but may be **long** if accessing them as such is more efficient. It is interesting that the word length, which is one of the machine differences that springs first to mind, has caused rather little trouble. A small amount of code (mostly concerned with output conversion) assumes a twos complement representation.

10.2 Unsigned Integers

Unsigned integers corresponding to **short** and **int** must be provided. The most relevant properties of unsigned integers appear when they are compared or serve as numerators in division and remaindering. Unsigned arithmetic may be somewhat expensive to implement on some machines, particularly if the number representation is sign-magnitude or ones complement. No use is made of unsigned **long** integers.

10.3 Characters

A representation of characters (bytes) must be provided with at least 8 bits per byte. It is irrelevant whether bytes are signed, as in the PDP-11, or not, as in all other known machines. It is moderately important that an integer of any kind be divisible evenly into bytes. Most programs make no explicit use of this fact, but the I/O system uses it heavily. (This tends to rule out one plausible representation of characters on the DEC PDP-10, which is able to access five 7-bit characters in a 36-bit word with one bit left over. Fortunately, that machine can access four 9-bit characters equally well.) Almost all programs are independent of the order in which the bytes making up an integer are stored, but see the discussion above on this issue.

A fair number of programs assume that the character set is ASCII. Usually the dependence is relatively minor, as when a character is tested for being a lower case letter by asking if it is between a and z (which is not a correct test in EBCDIC). Here the test could be easily replaced by a call to a standard macro. Other programs that use characters to index a table would be much more difficult to render insensitive to the character set. ASCII is, after all, a U. S. national standard; we are inclined to make it a UNIX standard as well, while not ruling out C compilers for other systems based on other character sets (in fact the current IBM System/370 compiler uses EBCDIC).

10.4 Pointers

Pointers to objects of the various basic types are used very heavily. Frequent operations on pointers include assignment, comparison, addition and subtraction of an integer, and dereferencing to yield the object to which the pointer points. It was frequently assumed in earlier UNIX code that pointers and integers had a similar representation (for example, that they occupied the same space). Now this assumption is no longer made in the programs that have been moved. Nevertheless, the representation of pointers remains very important, particularly in regard to character pointers, which are used freely. A word-addressed machine that lacks any natural representation of a character pointer may suffer serious inefficiency for some programs.

10.5 Functions and the calling sequence

UNIX programs tend to be built out of many small, frequently called functions. It is not unusual to find a program that spends 20 percent of its time in the function prologue and epilogue sequence, nor one in which 20 percent of the code is concerned with preparing function argument lists. On the PDP-11/70 the calling sequence is relatively efficient (it costs about 20 microseconds to call and return from a function) so it is clear that a less efficient calling sequence will be quite expensive. Any function in C may be recursive (without special declaration) and most possess several "automatic" variables local to each invocation. These characteristics suggest strongly that a stack must be used to store the automatic variables, caller's return point, and saved registers local to each function; in turn, the attractiveness of an implementation will depend heavily on the ease with which a stack can be maintained. Machines with too few index or base registers may not be able to support the language well.

Efficiency is important in designing a calling sequence; moreover, decisions made here tend to have wide implications. For example, some machines have a preferred direction of growth for the stack. On the PDP-11, the stack is practically forced to grow towards smaller addresses; on the Interdata the stack prefers (somewhat more weakly) to grow upwards. Differences in the direction of stack growth leads to differences in the operating system, as has already been mentioned.

XI. THE MACHINE MODEL OF UNIX

The definition of C suggests that some machines are more suitable for C implementations than others; likewise, the design of the UNIX kernel fits in well with some machine architectures and poorly with others. Once again, the requirements are not absolute, but a serious enough mismatch may make an implementation unattractive. Because the system is written in C, of course, a (perhaps necessarily) slow or bulky implementation of the language will lead to a slow or bulky operating system, so the remarks in the previous section apply. But other aspects of machine design are especially relevant to the operating system.

11.1 Mapping and the user program

As discussed in other papers, [18, 21] the system provides user programs with an address space consisting of up to three logical segments containing the program text, an extensible data region, and a stack. Since the stack and the data are both allowed to grow at one edge, it is desirable (especially where the virtual address space is limited) that one grow in the negative direction, towards the other, so as to optimize the use of the address space. A few programs still assume that the data space grows in the positive direction (so that an array at its end can grow contiguously), although we have tried to minimize this usage. If the virtual address space is large, there is little loss in allowing both the data and stack areas to grow upwards.

The PDP-11 and the Interdata provide examples of what can be done. On the former machine, the data area begins at the end of the program text and grows upwards, while the stack begins at the end of the virtual address space and grows downwards; this is, happily, the natural direction of growth for the stack. On the Interdata the data space begins after the program and grows upwards; the stack begins at a fixed location and also grows upwards. The layout provides for a stack of at most 128K bytes and a data area of 852K bytes less the program size, as compared to the total data and stack space of 64K bytes possible on the PDP-11.

It is hard to characterize precisely what is required of a memory mapping scheme except by discussing, as we do here, the uses to which it is put. In general, paging or segmentation schemes seem to offer sufficient generality to make implementation simple; a single base and limit register (or even dual registers, if it is desired to write-protect the program text) are marginal, because of the difficulty of providing independently growable data and stack areas.

11.2 Mapping and the kernel

When a process is running in the UNIX kernel, a fixed region of the kernel's address space contains data specific to that process, including its kernel stack. Switching processes essentially involves changing the address map so that the same fixed range of virtual addresses refers to the data area and stack of the new process. This implies, of course, that the kernel runs in mapped mode, so that mapping should not be tied to operating in user mode. It also means that if the machine has but a single set of mapping specification registers, these registers will have to be reloaded on each system call and certain interrupts, for example from the clock. This causes no logical problems but may affect efficiency.

11.3 Other considerations

Many other aspects of machine design are relevant to implementation of the operating system but are probably less important, because on most machines they are likely to cause no difficulty. Still, it is worthwhile to attempt a list.

(i) The machine must have a clock capable of generating interrupts at a rate not far from 50 or 60 Hz. The interrupts are used to schedule internal events such as delays for mechanical motion on typewriters. As written, the system uses clock interrupts to maintain absolute time, so the interrupt rate should be accurate in the long run. However, changes to consult a separate time-of-day clock would be minimal.

(ii) All disk devices should be able to handle the same, relatively small, block sizes. The current system usually reads and writes 512-byte blocks. This number is easy to change, but if it is made much larger, the efficacy of the system's cache scheme will degrade seriously unless a large amount of memory is devoted to buffers.

XII. WHAT HAS BEEN ACCOMPLISHED?

In about six months, we have been able to move the UNIX operating system and much of its software from its original host, the PDP11, to another, rather different machine, the Interdata 8/32. The standard of portability achieved is fairly high for such an ambitious project: the operating system (outside of device drivers and assembly language primitives) is about 95 percent unchanged between the two systems; inherently machine-dependent software such as the compiler, assembler, loader, and debugger are 75 to 80 percent unchanged; other user-level software (amounting to about 20,000 lines so far) is identical, with few exceptions, on the two machines.

It is true that moving a program from one machine to another does not guarantee that it can be moved to a third. There are many issues in portability about which we worried in a theoretical way without having to face them in fact. It would be interesting, for example, to tackle a machine in which pointers were a different size from integers, or in which character pointers were fundamentally different in structure from integer pointers, or with a different character set. There are probably even issues in portability that we failed to consider at all. Nevertheless, moving UNIX to a third new machine, or a fourth, will be easier than it was to the second. The operating system and the software have been carefully parameterized, and this will not have to be done again. We have also learned a great deal about the critical issues (the "hard parts").

There are deeper limitations to the generality of what we have done. Consider the use of memory mapping: if the hardware cannot support the model assumed by the code as it is written, the code must be changed. This may not be difficult, but it does represent a loss of portability. Correspondingly, the system as written does not take advantage of extra capability beyond its model, so it does not support (for example) demand paging. Again, this would require new code. More generally, algorithms do not always scale well; the optimal methods of sorting files of ten, a thousand, and a million elements do not much resemble one another. Likewise, some of the design of the system as it exists may have to be reworked to take full advantage of machines much more powerful (along many possible dimensions) than those for which it was designed. This seems to be an inherent limit to portability; it can only be handled by making the system easy to change, rather than easily portable unchanged. Although we believe UNIX possesses both virtues, only the latter is the subject of this paper.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
3. W. S. Brown, ALTRAN User's Manual, 4th ed., Murray Hill, N.J.,: Bell Laboratories, 1977.
4. B. G. Ryder, "The PFORT Verifier," Software - Practice and Experience, (October-December 1974), pp. 359-377.
5. American National Standard FORTRAN, New York, N.Y.: American National Standards Institute, 1966. (ANSI X3.9)
6. P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," AC M Trans. Math. Soft. (1978), to appear.
7. R. E. Griswold, J. Poage, and I. Polonsky, The SNOBOL4 Programming Language, Englewood Cliffs, N. J.: Prentice-Hall, 1971.
8. A. Snyder, A Portable Compiler for the Language C, Cambridge, Mass.: Master's Thesis, M.I.T., 1974.
9. D. Morris, G. R. Frank, and C. J. Theaker, "Machine-Independent Operating Systems," in Information Processing 77, North-Holland (1977), pp. 819-825.
10. J. E. Stoy and C. Strachey, "OS6--An experimental operating system for a small computer. Part 1: General principles and structure," Comp. J., 15 (May 1972), pp. 117-124.
11. J. E. Stoy and C. Strachey, "OS6--An experimental operating system for a small computer. Part 2: Input/output and filing system." Comp. J., 15 (August 1972), pp. 195-203.
12. 12. D. Thalmann and B. Levrat, "SPIP, a Way of Writing Portable Operating Systems," Proc. ACM Computing Symposium (1977), pp. 452-459.
13. L. S. Melen, A Portable Real-Time Executive, Thoth. Waterloo, Ontario, Canada: Master's Thesis, Dept. of Computer Science, University of Waterloo, October 1976.
14. T. L. Lyon, private communication
15. R. Miller, "UNIX - A Portable Operating System?" Australian Universities Computing Science Seminar (February, 1978).
16. R. Miller, private communication.
17. 17. S. C. Johnson, "A Portable Compiler: Theory and Practice," Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).
18. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
19. D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," B.S.T.J., this issue, pp. 1947-1969. Also in Proc. Hawaii International Conference on Systems Science, Honolulu, Hawaii, Jan. 1977.
20. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, "The C Programming Language," Comp. Sci. Tech. Rep. No. 31, Bell Laboratories (October 1975).
21. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," B.S.T.J., this issue, pp. 1931-1946.