A DESIGN FOR A PORTABLE PROGRAMMING SYSTEM

by

R. Braga
M. A. Malcolm
G. R. Sager

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Research Report CS-75-29

November 1975

A Design for a Portable Programming System

R. Braga
M. A. Malcolm
G. R. Sager

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

abstract: Our goal is to implement a complete system which can be ported easily to a new set of hardware. Whereas many components of the system can be implemented in a machine independent fashion, some of the most basic components (compiler, input/output control and file system) are necessarily machine dependent and must be ported. We therefore abstract these components and provide a set of tools to aid in the port. With the proper choice of abstractions and tools, the amount of work required to port the system is far less than that required to re-implement it. In this discussion, we focus mainly on the design of the compiler, loader and input/output components, and consider how they can be ported to an arbitrary machine.

1. introduction

As minicomputer hardware continues to decrease in cost and increase in speed and reliability, and software production costs remain relatively constant, software portability becomes increasingly important. Portability is a measure of the effort required to move a piece of software from one environment to another. A piece of software is said to be portable over a set of machine environments if it is significantly easier to port to a new environment than to reimplement. To be practical, it should not require much more effort to initially implement a portable version of a program than it takes to implement it in a highly machine-dependent form; and the portable version should not be prohibitively less efficient. For many programs, it is desirable to have portability-in-time as well as portability-in-space; that is, it is important to be able to port programs to new machines as they become announced, as well as to dissimilar existing machines. A machine-independent program is a portable program which requires no effort to port except that required to physically move the program to the new machine (which may be nontrivial, see Waite (1975)).

All existing techniques for writing portable programs entail programming in some abstraction of the machine environments over which portability is desired. A widely practiced technique, for example, is to program in a restricted subset of Fortran, often called PFORT (see Ryder (1974)), which is closely related to ANSI Standard Fortran. PFORT is a generally-available abstraction of medium to large-scale machines which is suitable for many applications. Another well-known technique is to devise an "abstract machine" appropriate for the application which can be implemented on a variety of machines. The abstract machine code may be executed interpretively to conserve core memory on small machines, or translated to the target computer's machine language using a portable macro processor or vendor-supplied macro assembler (see Colin, Shorey and Teasdale (1975)). These methods have been exploited by Waite and Poole (1973), Griswold (1972), and others.

Although the various methods for acheiving portability differ in approach, they share common shortcomings: The programming often has to be done in an awkward language; implementation of a portable program requires a great deal of skill; porting is often a task for the original implementor, if not for a highly skilled programmer;

the interface between the program and the various operating systems it must run under is usually crude at best, severely limiting the applications for which portable programming is possible.

Ideally, one would like to be able to write efficient programs in a nicely designed high-level language and have them be highly portable as a matter of coincidence. It should not require a herculean intellectual effort to produce a portable, or even machine-independent, program; rather, it should be difficult to avoid portability and easy to achieve machine independence. In this paper we present a design for a programming system which attempts to achieve such an ideal in the context of minicomputer applications.

Our system is based on a high-level language. We avoid the problems of interfacing with different operating systems by developing a complete portable system. In this way, only the compiler and part of the system need to be ported; most of the system and user programs will be machine independent. (Technically, the compiler is nearly machine-independent; but it must be bootstrapped from one machine environment to another.)

In the next section, we briefly discuss some of the reasons we've chosen to develop an entire system instead of (say) interfacing a portable compiler with various operating systems. In Section 3 we discuss the implementation language, and and its compiler. Section 4 contains a description of our highly portable linking loader whose major component is a machine-independent linking relocator which converts the relocatable compiler output to absolute executable format. In Section 5 we discuss environmental considerations, such as input/output, which are necessairly machine-dependent. Section 6 contains a description of the steps involved in various alternative ways of bootstrapping the system to a new machine; a new notation for describing such bootstraps is introduced and used in the discussion.

2. <u>why port the entire system?</u>

We have found considerable advantages in assuming there may be no pre-existing software for the target machine, in addition to the fact that some machines are marketed with essentially no software.

Inconsistencies in the kinds of operating system services available is a major motivation for producing a complete system. For example, a service as simple as receiving input from a terminal varies considerably from one system to the next. On some systems, input lines are padded with blank characters to make them a fixed length. On other systems, line delimiter characters are used. On some systems the input is automatically echoed back to the terminal; on others the user program must do the echo explicitly. Of course, even the character set and its internal representation varies from one system to another, and from one system release to another, and even within different parts of some systems. More complicated operating system services like tape or disc I/O or memory allocation have more serious inconsistencies involving character size vs. byte size, logical and physical blocking of records, etc.

One of the more difficult aspects of porting a compiler to various machines is its relationship to the pre-existing assembler and/or relocatable loader. Different systems have different restrictions on external symbols and linkage conventions. Library search rules and capabilities differ from one system to the next. Hence, serious development of portable libraries which must be used in a variety of operating environments is difficult since one cannot insure that external names remain unique, and that libraries are searched in the proper order.

Many of these standard problems of portable software can be avoided if one is willing to port the operating system along with language processors and user programs. That is, if much of a program's environment is ported with it, then many aspects of programs which are normally regarded as "machine-dependent" automatically become machine-independent. Of course, porting a system presents a number of new problems, many of which are difficult to solve. However, these problems must be addressed only by the persons who port the system, not by every user programmer who wants to write a portable program.

3. the systems implementation language

We have chosen the language B as a systems implementation language for a variety of reasons. Foremost among these is the fact that we have no desire to be sidetracked into the domain of language design at this point in the research. B was developed at Bell Laboratories. Along with its predecessor, BCPL (Richards (1969)), and successor, C (Ritchie (1974)), B has been extensively used as a systems implementation language. B is a simple block-structured language, well suited to systems programming. Like BCPL, B is a stack based language in which a program must be written as a collection of functions. The functions are dynamically nested, so that all program variables are either global to the whole program, or local to a particular function and dynamically allocated in the stack. B is typeless and word oriented; it contains a powerful set of operators. Functions can be recursive or reentrant.

The B compiler is well suited for our preliminary experimentation with portability because it is nicely structured and therefore easily modified to generate code for other machines. This is largely due to the fact that it is a syntax directed compiler for a language which has a simple and compact syntax. The one-pass compiler is implemented in B.

On the other hand, we feel there are several shortcomings to the language and compiler which need to be overcome in the future. Fortunately, it will be fairly easy to extend the syntax of B when the need arises.

One of the major defects we see in most languages is that they do not encourage the writing of machine-independent programs; most, in fact, seem to encourage machine dependence. Although we do not feel that it is possible to design a language which will prevent machine-dependent programming, we think it is possible to encourage it, both with the features of the language, and with a "verifier" which will scan a program for possible machine dependencies. Since we have control over both the system and the language, we hope to reduce the opportunities for machine dependence, and also provide a verifier which has a high probability of being able to detect it.

A question of prime importance is how to design the code generation modules of the compiler to make porting to a new machine relatively easy and the generated code very efficient, without assuming too much about the set of target machines. Our approach is to provide primitive functions in the compiler for emitting relocatable load code (discussed in the next section); and in the semantic routines, elementary machine-specific functions are called which use the code emitting functions. Coding these machine-specific functions is a major portion of the work involved in porting the system. They can be thought of as an abstract machine, which we call lowB; the semantic routines of the B compiler can be thought of as generating lowB which is mapped into relocatable load code by the machine-specific functions. Writing the lowB functions is entirely analogous to writing macros to map intermediate abstract machine language into machine language for the target machine. The amount of coding and debugging time required to write the lowB functions is roughly comparable to that required to implement the abstract machine SIL when porting SNOBOL (Griswold (1972)).

It is important to observe that although the lowB code generation functions are machine-specific, they are not machine-dependent. The compiler can execute on any machine to generate code for the target machine. In this way it can function as either a compiler on the target, or as a cross compiler on some other host machine.

4. __the__ __linking__ __loader__

A linking loader has three major functions: (1) the linking of modules via external symbols, (2) the relocation of addresses within modules, and (3) loading (and perhaps initiating) the executable module into memory. The relocation and loading steps may be combined, depending on the machine architecture. The three functions may be accomplished by one or more programs. The latter function is typically quite machine dependent.

If the loader is an abstract machine which takes the language processor output (load code) and executes it as a set of directives to build an absolute form of the executable program, then, with the proper set of directives, the loader could load for any machine architecture. These load code directives must describe to the loader the addressing structure and relocation procedures to be followed; this machine-specific information can be included as a standard "prelude" to each load code module generated by the compiler.

We have implemented the __Universal__ __Loader__ as two distinct phases; first, the linking relocator performs all linking and relocation, then outputs an absolute executable module which is loaded by the second phase, or __absolute__ __loader__. This two-phase operation is motivated largely by our desire for ease of portability: during a port, we must communicate executable programs to the target machine. This final communication step is easily accomplished by an absolute loader which can usually be coded in 20-40 instructions.

The linking relocator is written in B and forms the heart of the Universal Loader; we shall henceforth refer to it as "ULD". ULD makes two passes over the load code before generating the absolute module. This means that ULD need not reside in the same storage as the module it is building, and therefore does not impose any size restrictions. In the first pass, symbol tables are formed and the sizes of program modules are computed. In the second pass, external references are resolved, relocations are performed, and the absolute module is generated. The two pass operation avoids the problem of forming linked lists of unresolved references.

An absolute executable module is generated by ULD instead of a core image for two reasons: (1) The device used to load the program may be unreliable, therefore checksums are used for error detection; (2) the B compiler relies on a backplugging capability during code generation, and ULD doesn't resolve them since it is much easier to take care of with an absolute loader.

Another design feature of ULD is that it works with 8-bit bytes. In cases where the target machine has a word size which is not an even multiple of 8 bits, ULD will "waste" the extra bits of data bytes corresponding to the high order bits of words in the target.

The abstract machine which defines ULD has a directive register which will hold a 1- or 2-byte load directive and a working register which can hold up to 32 bytes of data. The load directives are used to:

- define an external symbol
- indicate a reference to an external symbol
- load data into the working register

- relocate fields in the working register
- alter descriptors to define fields to be relocated

During the first pass, only those directives which indicate the defining occurrence of an external symbol and those which write the working register need be interpreted. (The latter will indicate the number of bytes being written.) During the second pass, the directives which indicate the defining occurrence of an external symbol are ignored. Relocation is accomplished by first loading the working register, then issuing the directive to relocate. The relocate directive operates on data in the working register, a descriptor and a relocation constant. The descriptor contains the field size in bytes, the offset of the rightmost bit of the field relative to the rightmost bit in the byte, and a string of bytes used to mask out invariant bits within the affected bytes. Up to 8 different descriptors can be defined at any time. If more are needed, the compiler can redefine descriptors as many times as necessary during a load. The relocation constants are kept in the symbol table as strings of bytes, with the value right justified. In each module, up to 255 different relocation constants can be referenced. Relocation addition is performed as a byte-serial operation.

In order to clarify the role of the working register, descriptors and relocation constants, we will describe the relocation directive in more detail. The relocation directive (issued after a load working register directive) contains, for each relocation field, the following information:

1- the byte position of the leftmost bit of the field, relative to the beginning of the working register
2- the descriptor id

3- the relocation constant id

This information is encoded in a 2-byte field.

The task of defining descriptors is simple; we will show one example: if the target machine has a word size of 20 bits, and the address is located in the lower 10 bits, a target machine word would be defined as 3 bytes with the first byte having data only in the lower 4 bits. The descriptor for relocating such addresses contains:

- size: 2 bytes (the address will be in 2 adjacent bytes)
- mask bytes: 003 377 (octal notation)

If the address is in the upper 10 bits, then the descriptor would contain:

- size: 2 bytes
- mask: 017 374

Using the information in the relocation directive, we pinpoint the bytes in the working register which contain the address to be relocated, then from the descriptor determine the exact bits within the field which must be modified. The relocation constant id tells us which symbol table entry to use for the relocation. The write working register directive is issued after the data in the working register has been relocated; it causes the output of one record of data in absolute load code form:

1. leader of null bytes
2. rubout character
3. address bytes (typically 2 or 3)
4. byte count
5. data bytes
6. checksum or parity byte(s)

The exact form is varied to allow for ease of coding of the absolute loader and for the device characteristics. The leader and rubout are provided for ease of locating and positioning of the records in a paper tape reader. In the cases of magnetic tape or disc peripheral devices, the code for the write working register directive may be rewritten so that only parts 3 through 5 of the record are output.

5. <u>environmental considerations</u>

There are several important considerations for the port to the target machine: (1) we must certify that the appropriate parts of the compiler are properly recoded, (2) we must make provisions for the input/output requirements of the ported system components and (3) we must decide how much of the work is to be done on the host, and how much on the target.

In order to certify the implementation of the compiler, we intend to develop a set of diagnostic programs which test the machine independent features of the compiler required for successful operation of the system. These diagnostics will help locate bugs in freshly ported compilers and will aid the systems implementor in the correcting problems. They will be similar to a set of hardware diagnostics.

The diagnostic programs must be designed to be successively more complicated, since the more sophisticated diagnostics will require that a certain amount of the compiler be working properly. The first diagnostics will be so simple as to be compilable by an incomplete version of the compiler, thereby allowing the testing procedure to begin at an early stage.

Perhaps the most difficult environmental problems occur in input/output. At the highest level of abstraction, the programmer will see the input/output as completely byte oriented. The input/output routines will be much like those described in "The portable C library (on UNIX)", Lesk (1975). The programmmer is responsible for defining his own record blocking; as long as his routines are designed to agree on blocking, he will have no problems since the system will treat all data as binary bytes. Thus, the basic system does little for the programmmer, and (hopefully) nothing against him.

At the next lower level of abstraction, we must define the translations which make block devices behave like character devices. The routines which do this are easily written in B. Going lower still in the abstraction, we define processes which connect to the actual devices through some logical/physical mapping of files to devices, much as in UNIX (see Ritchie and Thompson (1974)). And at the lowest level, we need small machine-dependent modules to execute the input/output instructions and dispatch the processes connected to devices when interrupts occur. These routines would correspond to low.s, mch.s and conf.c in UNIX. Currently we have only primitive interactive tools for aiding the system implementer in creating load code modules for these routines. We are working on more convenient methods of preparing and maintaining these low-level modules.

6. bootstrapping notation

Before discussing the sequnce of events necessary to port the programming system to another machine, we introduce some notation to describe the operations involved. We may think of the important components of the system as transformations on their inputs. As an example, the operation of the compiler may be expressed:

CyEy: XS => XLy

where Cy is the compiler for machine y, with the suffix Ey to indicate that this version executes on machine y; X is an arbitrary program, with suffix S to indicate source and Ly to indicate load code for machine y. A cross compilation would be expressed as:

CtEh: XS => XLt

where the letters t and h indicate the target and host machines, respectively. In general, we may represent the operation of the compiler as:

CyEz: XS => XLy

where y=z for on site operation and y≠z for cross operation.

For ULD there is no need to indicate the orientation of the source, since it will load for any machine as it stands. However, the compiled and loaded version will execute only on a specific machine, thus:

UEh: XLy => XEy

where y=h for on site loading and y≠h for cross loads. We must stress that ULD will do both the on site and cross loads; the determining factor is the description of the target machine which is built in to the load code itself. In actuality, XEy is not directly executable, but must be initiated by the absolute loader described above. Since the absolute loader is a trivial transformation on its input, we do not consider it in our notation.

Continuing with the above development, we may consider a simple bootstrap sequence:

1. The source and object versions of system components exist on the host machine (h). In particular, the compiler (ChEh and ChS) and loader (UEh and US) must be ported to the target machine (t). The first step is to develop a description of t for ULD and design the load code to output for t by rewriting ChS to obtain CtS.
2. Compile CtS using ChEh; that is:

   ChEh(CtS) = CtLh.
3. Load the new version of the compiler:

   UEh(CtLh) = CtEh.

   We now have a cross compiler.
4. Continuing the port, we can recompile the compiler:

   CtEh(CtS) = CtLt

   and cross load it using ULD:

   UEh(CtLt) = CtEt.

   Alternatively, we could compile the loader, cross load it and load the compiler on site:

   CtEh(US) = ULt,

   UEh(ULt) = UEt,

   UEt(CtLt) = CtEt.

Obviously, the compiler is the most machine specific part of the system, and rewriting it will be the most time consuming part of the port. But we must also note that the other system components are not machine-specific, as is reflected in the notation: only the compiler requires two descriptors to indicate the machines involved. Once the compiler code generators have been rewritten, the possible paths by which we can move software to the target machine yield a large degree of flexibility.

We have neglected one important aspect of the problem; namely the support software required on the target machine. Most importantly, we must provide for the input/output necessary to operate ULD if we are to do on site loading. It is necessary to supply a certain amount of the support in the ported version itself. For the first test of the port, we have hand compiled, into load code, the two most primitive input/output operations needed by B as "busy-wait" input/output subroutines. For more sophisticated versions of the system, we are building a library of primitive operations on the input/output devices. A minor detail of the initial port is the implementation of an absolute loader. An example of one such absolute loader (for the Data General NOVA computer) is included in the Appendix.

7. conclusion

Our experiments in the development of a portable programming system have so far focused on the compiler and loader for the system. In this area, we have (1) adapted a local version of the B compiler to generate ULD code for the Microdata 1600/30 and (2) written ULD in B. As a cross-compiler for the Microdata 1600/30, the B compiler requires approximately 14,000 36-bit words of memory on the Honeywell 6050; we estimate that it will require a little less than 30,000 bytes of memory on the Microdata 1600/30 when it is bootstrapped there. The ULD program requires 3,000 36-bit words on the Honeywell, and 10,000 bytes on the Microdata. We have tested ULD by loading programs for the Honeywell 6050, Data General NOVA 2 and Microdata 1600/30. All test loads were performed using the 6050; load code for the 6050 and NOVA was generated by hand, while load code for the Microdata was generated by compilations of small B programs. The absolute modules for the NOVA and Microdata have been executed on those machines.

The internal structure of the three machines we have loaded shows a great deal of variation:

(1) Honeywell 6050: 36 bit word, word addressing, 18 bit address, the address may appear in the left or right half of the word
(2) NOVA: 16 bit word, word addressing, 8 or 15 bit address
(3) Microdata: 16 bit word, byte addressing, 8 or 15 bit address, variable-length instructions

In order to run programs on the NOVA and Microdata, it has been necessary to hand code modules to perform simple input/output functions and include these with the modules submitted to ULD. We have also developed a primitive interactive program which aids in constructing these load modules for a large class of computers. However, we believe there is much work to be done on software tools for developing input/output primitives for target machines.

In the near future, we plan to port a number of programs, including ULD, to the Microdata; this requires the implementation of a small "stand alone" operating system which will simulate the richer environment of a file system and a general input/output facility. The stand alone system will support a number of programs during the initial phase of the port while the system for the target machine is being configured.

Work is currently underway to (1) implement a library file editor for maintenance of load code libraries, (2) remove remaning machine dependencies from the B compiler for the Microdata and port it to the Microdata, (3) develop a general strategy for implementation of interrupt handling to form the basis for a portable input/output system, (4) design and implement a small portable operating system kernel in B, and (5) design a file system and its access mechanisms which can be ported to a large variety of machines and devices.

8. acknowledgements

We wish to thank Gary Stafford, Lawrence Melen and Manfred Young for their contributions to the design and construction of part of the software described above. This work has been supported in part by the National Research Council of Canada.

9. bibliography

Colin, A.J.T., K. Shorey and W. Teasdale (1975), The translation and interpretation of STAB-12. Software Practice and Experience, vol. 5, 123-138.

Griswold, R.E. (1972), The Macro Implementation of SNOBOL4. W.H. Freeman & Co., San Francisco.

Lesk, M.E. (1975), The portable C library (on UNIX). Documents for use with the UNIX time-sharing system, sixth edition, Bell Laboratories, Murray Hill, New Jersey.

Richards, M. (1969), BCPL: a tool for compiler writing and system programming. Proc. Spring Joint Computer Conf., 557-566.

Ritchie, D.M. (1974), C reference manual. Bell Laboratories TM-74-1273-1.

Poole, P.C. and W.M. Waite (1973), Portability and adaptability. Lecture Notes on Econ. and Mathematical Systems, no. 81: Advanced Course on Software Engineering, Springer-Verlag, 183-277.

Ritchie, D.M. and K. Thompson (1974), The UNIX time-sharing system. Comm. A.C.M., vol 17, no 7, 365-375.

Ryder, B.G. (1974), The PFORT verifier. Software Practice and Experience, vol 4, no 4, 359-377.

Waite, W.M. (1975), Hints on distributing portable software. Software Practice and Experience, vol. 5, 295-308.

10. appendix: Absolute loader for the NOVA computer

The following NOVA program will read the output of ULD from a teletype and load it into memory. The format of the records is decribed in Section 4. The program dedicates register 1 to accumulate the sum of bytes, 2 to return input words, and 3 for subroutine linkage.

```
; ABSOLUTE LOADER FOR ULD OUTPUT
        .NREL
        NIOS    TTI      ; START THE INPUT
NEXT:   JSR     GETBY    ; BURN INTER-RECORD NULLS
        MOV     0,0,SNR
        JMP     NEXT
        SUB     1,1      ; ZERO AC1 (BYTE SUM)
        JSR     GETWD    ; GET THE LOAD ADDRESS
        STA     0,LOCN   ; SAVE FOR STUFFING DATA
        JSR     GETBY    ; GET BYTE COUNT
        MOVR    0,0,SNR  ; CONVERT TO WORD COUNT (START ADDRESS?)
        JMP     @LOCN    ; START THE PROGRAM
        STA     0,WDCNT  ; ELSE SAVE COUNT
READ:   JSR     GETWD    ; GET A DATA WORD
        STA     0,@LOCN  ; STUFF THE DATA WORD
        ISZ     LOCN     ; BUMP LOAD ADDRESS
        DSZ     WDCNT    ; DECREMENT WORD COUNT (LAST?)
        JMP     READ     ; DO IT SOME MORE
        STA     1,WDCNT  ; SAVE THE SUM OF BYTES
        JSR     GETWD    ; GET CHECKSUM
        LDA     1,WDCNT  ; COMPARE THE RESULTS
        SUB     0,1,SNR
        JMP     NEXT     ; GET NEXT RECORD
        HALT             ; CHECKSUM ERROR
;*** GET A WORD OF DATA ***
GETWD:  STA     3,RTRN   ; SAVE RETURN ADDRESS
        JSR     GETBY    ; GET FIRST BYTE
        MOVS    0,2      ; SWAP HALVES INTO AC2
        JSR     GETBY    ; GET SECOND BYTE
        ADD     2,0      ; STICK BYTES TOGETHER
        JMP     @RTRN
;*** GET A BYTE OF DATA ***
GETBY:  SKPDN   TTI
        JMP     .-1      ; WAIT FOR COMPLETION
        DIAS    0,TTI    ; GET THIS ONE AND START NEXT
        ADD     0,1      ; ACCUMULATE BYTE SUM
        JMP     0,3      ; RETURN
WDCNT:  0
RTRN:   0
LOCN:   0
        .END
```