Transporting Up: A Case Study

Victor R. Basili
Computer Science Department, University of Maryland

John G. Perry, Jr.
Naval Surface Weapons Center, Dahlgren Laboratory, Virginia

This paper discusses the various aspects of transporting one language, SIMPL-T, from the UNIVAC 1108 computer to the CDC 6700 computer, a machine of larger word size. Special attention is given to the operational mechanism, the testing plan, the problems encountered in transporting, and the various statistics kept during the development.

INTRODUCTION

It is the authors' experience that developing a transportable and efficient software product is a difficult task. The main difficulties arise in achieving a good transportable design of the product and in making the product operational. Design is difficult because of little theory for developing a transportable design applicable to a set of heterogeneous machines; also, a transportable design may conflict with other requirements, such as efficiency. Operational problems arise in making software that executes on one machine execute on another, due to the different machine environments, especially the different operating systems, as well as the testing requirement that the program perform the "same" on the new machine, to whatever degree that term is meant.

The specific software discussed is a compiler for the language SIMPL-T [1], a structured programming language in the FORTRAN/ALGOL 60 size range, and the project involves the bootstrapping of the compiler. The original compiler was developed for the UNIVAC 1100 series computer and was designed to be transportable and efficient, i.e., as efficient as the standard UNIVAC FORTRAN compiler [2]. Since the target architectures in mind had a smaller word size than the 36-bit-word UNIVAC machine, e.g., the IBM 360/370 and 16-bit minicomputers, it was assumed that bootstrapping to a machine with larger words would be easier; thus the problems of transporting up had not been thoroughly considered.

Address correspondence to Dr. Victor R. Basili, Computer Science Department, University of Maryland, College Park, Maryland 20742.

METHODS FOR TRANSPORTING

There are two possible approaches for transporting a compiler from one machine to another.

Standard Approach

The code generator on the host computer is modified to produce code for the target computer. Assuming the compiler for a language is written in the language itself, the source of the modified compiler can then be compiled on the host computer to yield a compiler executable on the target machine. A set of interface routines with the operating system that are not part of the compiler but are essential to the use of the compiler must be written. These interface routines (also called run-time routines or language-support routines) are usually written in a host machine language. Transporting a compiler involves three steps: (1) the development of the target compiler at the host site, (2) the development of the run-time routines at the target site, and (3) the installation and integration of the target compiler and run-time routines at the target site. Testing and debugging may require repeated use of the host machine, however.

Transportable Approach

The host computer's compiler is translated into another language that runs on the target computer. This creates a source for the host compiler that can operate on the target machine. It assumes the existence of a translator program that runs on the target machine that will convert a program to a high-level target language along with a set of run-time routines written in that language. For SIMPL-T, there exists such a translator written in SNOBOL 4 that translates SIMPL-T programs into ANSI standard FORTRAN, along with a set of FORTRAN run-time routines [2]. The benefit of this approach is that it does not require the use of the host computer [3].

TRANSPORTING AND TESTING

We shall now discuss the transporting of the SIMPL-T compiler to the CDC 6700 [4], a task that was completed in 5 person-months of effort.

The initial plan was to use the transportable bootstrap approach since the tools (e.g., the translator, FORTRAN routines) were available and had been used to bootstrap the compiler to an IBM 360/370 series computer. Also, the two sites were 60 miles apart and had no mutual teleprocessing capability. The transportable approach, however, proved unsatisfactory because the translator was written in SNOBOL 4 and a version with the necessary functions (SNOBOL 4 and at least 36-bit integer arithmetic) was not available on the CDC 6700. Thus, we were forced to the standard approach.

The approach will be discussed in two phases: the work predominantly done at the UNIVAC site and the work done at the CDC site. In phase 1 (at the UNIVAC site) the code generator of the host computer's compiler was redesigned to take into consideration the run-time environment of the CDC 6700. CDC assembly language (COMPASS), rather than machine code, was used as the target language in order to keep the project as uncomplicated as possible. COMPASS was selected since (1) relocatable binary is not easy to generate for the CDC system; (2) conversion (including character sets) to different CDC operating systems (SCOPE 3.3 and SCOPE 3.4) is easier; (3) the COMPASS source file could be changed on the CDC system to correct a code generator error or to change a test problem without having to return to the UNIVAC 1108; and (4) COMPASS code output of the code generator is more readable than machine code output.

About 40 SIMPL-T routines, written to check out the original compiler, were run on the UNIVAC 1108 to test the COMPASS code produced by the modified code generator. The checking was done visually, first checking the SIMPL-T code generator source (desk checking) and then checking the output (COMPASS) of the test cases produced by the new compiler. Finally, the SIMPL-T compiler was used as a test program to get a version of the compiler that would execute on the CDC machine.

The second phase was accomplished at the CDC 6700 site. The run-time support routines were written in COMPASS for the CDC machine. These were the routines (e.g., string packages, input/output, etc.) that enabled the new compiler to be integrated successfully with the target computer's operating system.

A check was made of the new code generator in its new environment, after the new compiler was compiled for the first time on the target machine's compiler. The check was performed by comparing the compiler output from the host compiler against the output of the target machine compiler. To check the CDC run-time routines, test routines written in SIMPL-T were used to (1) permit the use of the set of test routines already developed in SIMPL-T for the UNIVAC compiler effort; (2) minimize the time required to write new test routines; and (3) help check out the CDC code generator further. Each SIMPL-T test routine was used to test a particular CDC run-time routine; then if an error occurred during testing, it was assumed to be in that run-time routine. The new test routines were then added to the existing library of SIMPL-T test routines that could be used in testing modifications to the compiler being tested or any other bootstrapping projects involving the SIMPL-T language. The resulting group of test routines forms a standard that is used to help check that the various compilers are operating the "same."

There were two disadvantages to this SIMPL-T test program approach for the run-time routines. First, it was assumed that the code generator was error-free; therefore any error caught by the run-time test routines was in the run-time routine itself. If this were not the case, time was lost examining a good run-time routine. Second, if a test routine had not been designed well enough to test out all aspects of the routine, or the test needed to be enhanced to check out some newly exposed, potentially unreliable code segment more fully, or if an error of some kind were made, then recompilation was required. Recompilation implied returning to the UNIVAC site, making the change, recompiling, and then repeating the run on the CDC machine. Fortunately, since COMPASS rather than machine language was used as the target language, the COMPASS version of the SIMPL-T test program could be modified by using an editor to change or add new COMPASS statements to the test program. These changes could be accumulated, put into the SIMPL-T source test programs, and recompiled at one time. However, these local patches were possible for only minor changes in the COMPASS code; any serious changes required recompilation.

The final test program was the SIMPL-T compiler itself. The compiler consists of about 6000 SIMPL-T statements and produces about 100,000 lines of COMPASS code. The COMPASS code was then assembled and run.

TRANSPORTABILITY DESIGN PROBLEMS

As stated in the introduction, designing a transportable and efficient software product is a difficult task. First of all, there is little theory for the transportable design of software that is effective with respect to a set of time and storage efficiency constraints. This is not because all problems are hard to solve on a case-by-case basis: Each problem often has a reasonable solution [5]. The difficulties occur due to the diversity of problems and their solutions, which tend to be ad hoc and based on incomplete knowledge. Local decisions in one situation appear to be counterproductive to decisions made in another situation. The trade-offs among time, space, and portability appear to complicate the process considerably.

Mills [6] speaks about solving problems that are yet too hard; i.e., we have not yet enough understanding of a rigorous solution to a given problem and must use ad hoc methods or heuristics in specifying the solution. Developing a transportable product is just such a problem. The characteristics of such a product are not well defined except by the success of the final product to execute on several machine architectures.

Second, unlike efficiency, the final target for a transportable design is not very specific. It is not a unique machine with a given set of characteristics. It involves a multitude of architectures with a variety of special characteristics. For example, one may know how to write an efficient program on the UNIVAC 1108 because one understands the machine architecture; but there are too many variables involved in the attempt to develop a product to be efficient on a variety of machine architectures. That is, one cannot have them all in mind unless one distills away all of their idiosyncratic properties, leaving portability without efficiency. These variables complicate the product development process considerably.

Third, developing a product that is correct, reliable, and modifiable is difficult enough; adding a set of efficiency constraints complicates the problem further; and adding portability constraints raises the complexity of design one more level. The design level becomes more complex because several of the goals of one of these different product characteristics conflict with the goals of others. Thus, for example, a product that is efficient is often nonportable specifically because design decisions were made in favor of efficiency over portability. Also, portability introduces a new and complicated balance to the set of product properties.

The UNIVAC SIMPL-T compiler is very reliable: There has been success with several modifications; it generates efficient object code comparable with compilers for languages of its size; and there has been success with transporting the compiler onto other machines [1, 7, 8]. To support transportability, both the language and the compiler were designed with that goal in mind; the compiler having been written in the language.

The language contains character strings to minimize word size problems in handling textual data. Although there are several word-size-dependent operators in the language, such as bit, partword, and shift operators, their machine-dependent effects can be minimized without affecting efficiency by carefully using the parameterized define facility and macro preprocessor provided with the SIMPL system.

The front end of the SIMPL-T compiler (scanner and parser functions) has been designed to be machine independent. It interfaces to a machine-dependent code generator by generating a file of very-high-level, machine-independent quadruples (quads) transcending the architecture of any particular machine. (A quad is a 4-tuple of pseudocode that consists of an operation field, two operand fields when needed, and a result field when needed.) To minimize the word-size dependency problems, the compiler uses variable length character strings to represent textual data. All bit strings of information are packaged into multiples of some "transportable" word size (16 bits in this case) and machine-dependent operators are used sparingly and carefully in parameterized macros. To handle the machine dependency of internal character representations, 8-bit ASCII is used as the internal form.

Although transportability was designed into the compiler, which was transported successfully to smaller word-size machines, several problems arose with transporting to the CDC 6700. The UNIVAC front end was designed to handle a maximum of 36-bit integers, and 60-bit integers were required for the CDC 6700. Thus, checking for the size of constants in the conversion routines allowed a maximum integer of 2^{35}. This was clearly a problem on a machine that permitted a maximum of 2^{59}. This problem was solved by generating two extra versions of the compiler. A version was generated that did not check for maximum integer size. The modified compiler was then run through this nonchecking version and the checking was put back in, this time using a 60-bit constant for confirmation. The fact that the arithmetic of both computers was one's complement, so that integers were bit-for-bit compatible with the CDC integer, simplified the conversion effort. However, if the target machine were two's complement, extra checking and conversion would have been needed to convert from one's complement to two's complement. If a 60-bit arithmetic version of the SNOBOL 4 had been found, this problem would have been easily solvable since the change of constant could have been made in the SNOBOL translator itself.

A second problem was that the compiler did not make full use of the larger 60-bit word on the CDC machine. Information was stored in packets of 16 bits. Unfortunately, the macros were designed to handle 16- and 32-bit words easily, but multiples of three were not so easily handled. Thus the SIMPL-T compiler symbol table uses only the right 32-bits of the CDC's 60-bit word. One solution is to redefine the symbol table macros to use 48 of the 60 bits. This approach is feasible, but some work is involved since routines do not always use consistent access methods. More centralized symbol table access routines would have avoided this problem and isolated machine dependencies better.

A third problem involves the partword operators, operators that allow access of bit fields within a word. First, only 36 bits of the 60-bit CDC word are accessible because of error checks built into the UNIVAC compiler. Second, the SIMPL-T language defines the bit ordering from left to right; i.e., the left-most bit (the sign bit) is bit number zero. Thus a SIMPL-T program referencing bit zero would refer to an entirely different bit on the two machines; e.g., bit zero on the UNIVAC machine is 2^{35}, whereas while bit zero on the CDC machine is 2^{59}. The bit numbering from left to right was chosen to be compatible with the UNIVAC system use architecture. It permitted an efficient mapping into the machine defined bit/partword operators (e.g., quarter-word, half-word).

Here efficiency was considered above transportability since partword was defined in SIMPL-T specifically for efficiency. Partword operators were not transportable and it was not worth the effort to make them "pseudotransportable." They were to be made as efficient for the machine on which they were operating as possible, and any code written using them was to be rewritten for the new machine if that code was to be transported.

To handle the problem of transporting the partword code, the partword macros were redefined changing the maximums used in the range checks. Reversing the ordering of the bits to be more compatible with the CDC architecture standards was considered but not implemented in order to maintain compatibility with the UNIVAC version.

Character sets are traditionally a problem in bootstrapping whether moving to a larger or smaller machine. Although the SIMPL-T compiler uses 8-bit ASCII for its internal representation of characters, it lives with a character set of less than 2^6 characters. The number of characters permitted the mapping of the 8-bit ASCII code into the 6-bit display code available on the CDC machine. The mapping was one-to-one except for two characters which had to be changed. The only drawback to the character set mapping is that an extra conversion is needed, for example, to access into the ASCII hash coded symbol table. Unfortunately, this is an unavoidable problem if lexigraphic order is to be kept consistent from compiler to compiler.

STATISTICS ON THE PROJECT

Detailed information was kept on a daily basis for each stage of the project. Difficulties occurred in trying to define discrete stages (e.g., design, code, checkout) and being able to break down the time expenditures of each day into these categories. The terminology involved is not standardized; therefore, the discrete categories used are defined below.

Discussion: discussing global aspects of project and method used to develop the SIMPL-T UNIVAC compiler.

Background: researching reference material; writing special purpose programs to display the internal workings of the compiler (on the UNIVAC machine); getting the necessary background knowledge and studying source listings for the scanner and parser.

Design: designing, flowcharting, and outlining.

Redesign: redesigning a part of the project because the initial design was incomplete.

Code: writing on coding sheets, also including some minor design.

Check: visually checking logic of code; desk checking.

Debug: detecting, locating, isolating, and eliminating mistakes, malfunctions, or faults.

Test: writing or running test cases; also visually analyzing the results.

Keypunch: keypunching or entering code into a terminal.

Setup: preparing files, tapes, and jobs; including creating bootstrap tapes on the UNIVAC machine and transporting them into the CDC file system.

System Changes: making changes required by external system or compiler changes.

System Problems: making corrections or redoing work as a result of a system malfunction (software or hardware).

The statistics for the project are given for the UNIVAC (first) phase, the CDC (second) phase, and the total project. The steps were also grouped together to give percentages for the general areas of (1) design (discussion, background, design, and redesign); (2) code; (3) test (checking, debugging, and testing); and (4) miscellaneous (keypunch, setup, system changes, and system problems).

Table 1 gives a percentage breakdown of work on the project over nine calendar months (150 days or 746 hours). Only hours spent on the project are considered.

UNIVAC phase. The UNIVAC phase involved the general design of the code generator, the design of the CDC run-time environment, and the coding and testing on the UNIVAC machine of the CDC code generator. Note that the design percentage of the UNIVAC phase was kept to a minimum due to the simple nonoptimizing design of the code generator. If a more sophisticated code generator for the CDC compiler had been written, it would have required more design time.

CDC phase. The CDC phase involved the design, coding, and testing of about 50 independent and small run-time routines. It also included the final testing and actual bootstrapping of the compiler.

Table 1. Percentage of Time

	Phase 1	Phase 2	Total
Design	31	14	22
Discussion	4	3	3
Background	12	8	10
Design	10	3	6
Redesign	5	0	3
Coding	17	29	23
Testing	39	50	45
Check	6	6	6
Debug	10	12	11
Test	23	32	28
Miscellaneous	13	7	10
Keypunch	5	0	2
Setup	2	6	4
Changes	4	0	2
Problems	2	1	2

The reasons for the differences in the design phase percentages for the two phases are not surprising, since the second phase required very little design for the single-function, well-defined (algorithms existed for the UNIVAC compiler) run-time routines. Both coding and testing for the CDC phase were greater because of the large number of routines that needed to be written and checked. Most of the difference in the miscellaneous group was due to the fact that professional keypunching services were not available at the UNIVAC site.

It is believed that if an optimized code generator that produced relocatable code (as exists for the UNIVAC compiler) were written for the CDC compiler, then the percentages would be changed to approximately 35% for design, 15% for coding, 40% for testing, and 10% for miscellaneous.

Some productivity figures in the standard measures of lines/person-year and words/person-year are given in Table 2. Although these measures of productivity are not too meaningful and are rather ambiguous, they will permit us to compare our results with several contemporary "hypotheses" [9]:

1. Programmers produce a fixed number of statements per year, regardless of the language.
2. Programmer productivity is increased several fold when a suitable high-level language is used.

Before we make this comparison, let us qualify some aspects of the project that affect the productivity results to aid in interpretation of the results. The compiler generated in this project is effective but does not generate very efficient code; more work is needed to get a good optimizing compiler. There was only one programmer, working part time in a time-sharing environment at both sites. The majority of his experience was with the CDC environment. Coding was done partly in the assembly language COMPASS (with which he was familiar) and the structured programming language SIMPL-T (with which he was less familiar). The problem was a typical translator problem and was reasonably well specified. All code was written from scratch and almost no new documentation was required for the project.

Now let us examine the aforementioned "hypotheses" with respect to this project. The first statement does not appear to be supported by this project study if we compare the figures 22 vs 70 statements/day. However, the following two points tend to cloud the issue. First, this project used a computer with a primitive instruction set (CDC 6700), where, for example, three instructions are necessary to transfer

Table 2. Productivity

	Source language	Statement language	Statements[a]	Lines[b]	Words[c]	Hours[d]	Statements per hour	Statements per day
Phase 1	SIMPL-T	SIMPL-T	1,000[e]	1,700	9,536	353	2.8	22
	SIMPL-T	COMPASS	18,606	20,007	9,536	353	53.0	424
Phase 2	COMPASS	COMPASS	3,500	5,100	3,836	393	8.8	70

[a]Includes only lines with executable statements (no comment lines, no declarations, and no initialization; includes only statements contained in the code statements) written for the code generator and CDC run-time routines (no statements from test cases).

[b]Includes comments, declarations, etc.

[c]CDC 60-bit words (includes data storage).

[d]Total project hours: includes time spent on all aspects of the project (all 12 components considered in the separate time estimates—see Table 1).

[e]The SIMPL-T code generator produced COMPASS statements. These COMPASS statements (produced by the 1000 SIMPL-T source statements) are described by the second line of the table. The statistics here are a function of the quality of the code generator and the machine instruction set of the CDC 6700.

data from one memory location to another. Second, phase 2 of the project was composed of less complicated tasks than was phase 1; e.g., there were 50 small self-contained routines written in phase 2 as opposed to the one more complex code generator written in phase 1.

The second "hypothesis," that programmer productivity is increased by use of a high-level language, seems to be validated if we compare 424 statements/day, which is the number of COMPASS statements produced as output by the 1000 SIMPL-T statements, with 70 statements/day, which is the number of statements produced directly in COMPASS. This yields a 6:1 ratio.

Statistics were also kept on the number of errors uncovered in testing (Table 3). There were also four errors found in the final integration testing of phases 1 and 2. It is interesting to note that the ratio of statements written to errors found appears to be relatively independent of the source language; i.e., there was one error per 67 SIMPL-T statements and 70 COMPASS statements. If the error ratio can be shown to be consistent for individual programmers, one could show that the use of a high-order language reduces the words of storage generated per error ratio, just as it increases productivity; i.e., in this case, 636 words/error for SIMPL-T-generated code vs 77 words/error for COMPASS-generated code. Of course, this ratio cannot be taken too literally as a more efficient compiler should reduce the total number of words of COMPASS code generated by the CDC SIMPL-T compiler.

CONCLUSION

Based on this experience, we make several recommendations. The standard bootstrapping approach is fraught with operational problems that waste time and add the expense of extra knowledge requirements, e.g., an extra operating system.

Duplicate test cases for both versions of the compiler helps guarantee the "sameness" of both implementations. The same front end for both compilers also helps, especially with respect to the "sameness" of error messages and error recovery. However, nothing beats careful code reading.

There is software that is portable and efficient in the context of two or three machine architectures; it is rare and difficult to achieve portability and efficiency among many machine architectures. We suspect that small design problems will continue to plague the development of a transportable and efficient design without the development of a more rigorous theory. However, it is helpful to keep a checklist of the problems likely to be encountered in order

Table 3. Errors

	Source language	Statements	Words	Errors	Statements/error	Statements/error
Phase 1	SIMPL-T	1000	9536	15	67	636
Phase 2	COMPASS	3500	3836	50	70	77

to examine them in the context of new problems as they arise. The integer-size problem encountered in the existing design could have been solved using the portable bootstrap with maximum constants parameterized. The real problem is that it is not clear what new problems may be encountered with the existing design for some yet untried architecture.

Based on the error statistics and an analysis of the kinds of error committed, it is believed that code reading and walk-throughs would have greatly minimized the error rate and time. This would have required a second programmer being assigned to the project—an impossible situation in our case, but one for which we shall try to make arrangements in our next product development. Much has been said about having too many people on a project; not enough has been said about having too few—not from the point of view of project size but for the express purpose of reading each other's code. The authors believe two heads are better than one and that programming should not be an individual but a group activity.

We should like to see more statistics on error rate per lines of code written. It would be interesting to see if this is an invariant dependent upon the programmer or the type of project, as well as to see if other studies come up with a rate invariant across language level. If so, it would be another argument for high-level languages. Also, it would be interesting to see whether specific program development methodologies improve that rate on an individual basis. We should like to see more such statistics published in open literature.

REFERENCES

1. V. R. Basili and A. J. Turner, SIMPL-T, *A Structured Programming Language,* Paladin House, Geneva, Illinois, 1976.
2. V. R. Basili and A. J. Turner, A Transportable, Extendable Compiler, *Software Practice and Experience* 5, 269–278 (1975).
3. R. Dunn, SNOBOL 4 as a Language for Bootstrapping a Compiler, *SIGPLAN Notices* 8 (5), 28–32 (1973).
4. Control Data Corporation (CDC), *6400/6500/6600 Computer Reference Manual,* Publ. No. 60100000, St. Paul, Minnesota, 1969.
5. P. J. Brown, *Macro Processors and Techniques for Portable Software,* Wiley, New York, 1974.
6. H. D. Mills, Software Development, *Proc. 2nd Int. Conf. on Software Engineering,* October, 1976.
7. V. R. Basili, The Design and Implementation of a Family of Application-Oriented Languages, *Proc. 5th Texas Conf. on Computing Systems,* University of Texas, October 18–19, 1976, pp. 6–12.
8. K. Harada and V. R. Basili, Structured Programming Language for Complier Writing: SIMPL-T (in Japanese). *Information Processing* 17 (3), 222–228 (1976), Information Processing Society of Japan.
9. F. P. Brooks, *The Mythical Man-Month,* Addison-Wesley, Reading, Massachusetts, 1975.