UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Porting The Zed Compiler

G. Bert Bonkowski
W. Morven Gentleman
Michael A. Malcolm

May 1979
CS-79-22

PORTING THE ZED COMPILER

G. Bert Bonkowski

W. Morven Gentleman

Michael A. Malcolm

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

May 1979

Authors' addresses: G.B. Bonkowski, W.M. Gentleman and M.A. Malcolm, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

Porting The Zed Compiler

G. Bert Bonkowski
W. Morven Gentleman
Michael A. Malcolm

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

Introduction and Context

Zed is the base language used to implement the portable realtime operating system Thoth[7], and to write commands, utilities, application programs, and other software which run under Thoth. (Zed is similar to C, although language details are not important in this paper.) One of the founding principles of Thoth is our experience that the hardest problems in porting programs usually arise when interfacing to different operating systems. By porting the whole operating system first, we ensure that other programs see the same system interface. Hence, with a modicum of care in programming, programs can be machine independent in that, without change, they can be compiled for any machine to which Thoth has been ported. Since the Zed compiler is a program which runs under the Thoth operating system, it is portable in this sense, and the same source serves to produce either native mode or cross compilers because it does not depend on the machine on which the compiler will run.

Compilers, in common with certain other "system software", are, however, machine specific in that the output from executing a compiler must be different depending upon the machine on which that output is destined to be run. Clearly, for Thoth to be ported to many machines, the compiler should be portable in this second sense: as much as possible of the source of the compiler should be machine invariant, that is, independent of the target machine, and for those parts which are machine specific, there must be a well organized scheme to assist realizing versions for new machines. The Zed compiler is designed to ease such ports. Actually, the objective is broader than that, because the initial port is only part of the problem: once versions of the compiler are available for several machines, we still have the problems of maintaining all these different versions, enhancing them, and evolving them to follow changes in the language. To do this, we need a structure that is easy to understand and to work with. We also want to minimize the amount of source code for which multiple versions must exist.

The purpose of this paper is to illustrate how, by careful attention to the structure of the compiler, its decomposition into modules, the definitions of the interfaces between those modules, and the data structures and algorithms it uses, a fairly conventional[10,21] production quality compiler can provide this kind of portability. The structure finally chosen has other virtues - of the many compilers we have written, this one is the easiest to learn and to work on, so we will use this structure again for other languages. We will describe the structure by outlining what must be done to port the compiler to a new machine.

It has been our experience that with adequate understanding of the abstractions of phases, algorithms, data structures, etc. in compilers, these entities can be machine invariant at some level. However, the design of the generated code, i.e. of what code bursts to produce corresponding to each source language construct, is a much more sensitive matter. We are concerned with the production of very high quality code on many diverse architectures. Since Thoth is a realtime operating system which must run even on small minicomputers, execution speed is important but the size of generated code and the sizes of data structures such as stack frames are even more important. (The compiler itself must also run on minicomputers, and consequently some of the structuring issues involve how to keep its size down.) Since this is a production compiler, rather than an academic research vehicle, we are concerned with the quality of code in a statistical sense, not with respect to isolated case studies. It is important that things which actually are done often (either in a static or dynamic sense) be done well - things that are done rarely do not matter. We have over 70,000 lines of Zed source which we can study. We want the ompiler writer to have the tools and facilities to concentrate on the creative task of best exploiting the instruction set.

Significantly, we are not trying to automate compiler construction. In the past few years, there have been several attempts to automate code generators, starting from some description of the instruction set[5,6,9,14]. We are very pessimistic about such a process achieving the quality of code we desire except for the simplest and most regular machines, such as the P-code interpreter or perhaps the IBM 370 or PDP 11. This pessimism is based on our experience that even for machines we have known for as much as 15 years, improvements in generated code of 5 to 10 percent still occur. Our view finds support even amongst people who have attempted automatic derivation of code generators. For instance, at the end of Glanville's thesis[9], he remarks that his scheme does not work well with rich but irregular instruction sets. Given the objective of portability, the existence of irregular instruction sets must be accepted as a fact of life. Machines with irregular instruction sets will continue to be built - geometry constraints and real estate limitations in LSI fabrication introduce irregularity into microprocessors, for instance. Furthermore, irregularity may not be undesirable. Tanenbaum[19] and others have observed that an instruction set which provides appropriate special cases and "Huffman encodes" constructs that happen frequently can provide substantially more compact code (with corresponding direct and indirect performance advantages) than a fully orthogonal and regular instruction set. Machines with such beneficial irregularity exist now, and our approach has little difficulty compiling for them.

Strategy

The design of the Zed compiler differs from that of other portable compilers[1,8,13,16] in three major ways, and these issues will be examined before describing the compiler in detail. The first difference is that we chose to generate relocatable object code rather than assembly code. The second difference is that we chose to implement a multi-pass rather than a single pass compiler. The third difference is that we chose to represent the source program in an intermediate language which is tree based, rather than with tuples or as source for a simple stack machine.

Compilers often generate source for an assembler rather than directly producing source for a loader. The motivations for this are usually fourfold: assembler source can be read by the compiler writer to assist in debugging the compiler; assembler source may be simpler to produce than loader source; the compiler writer may want to avoid having an extra pass to do backplugging or to resolve whether to use short or long form jumps, and he may be able to get the assembler, but not the loader, to do this for him; for portable compilers, the fact that source for existing assemblers is more uniform than source for existing loaders may make it a preferable interface. These motivations do not apply in the context of a compiler running under Thoth. We invested in developing a portable disassembler, so we do not need assembler source for debugging. The loader is part of the Thoth environment[17], and hence under our control, so we can make it easier to feed than an assembler, and smart enough to do whatever we need - and we made the input to the loader machine invariant.

We have a number of other reasons for wanting to generate load code directly. The first is speed. An assembler phase is slow, if only because of the additional text I/O which must be done. Second, using an assembler phase requires a good enough assembler - and while we do implement an assembler during a port, we do not want to be obliged to implement a very clever one. Third, the format of assembler source, especially for elaborate addressing modes, may be harder to generate than the actual bit patterns of instruction fields. Fourth, there are problems of names and scopes associated with the language feature of inserting machine instructions inline. Fifth, we want to facilitate separate compilation by having the compiler access libraries to check for consistent usage, and we do not want it to parse source to do it.

The primary motivation for having a multi-pass compiler is that it reduces the size of any given pass, increasing the ability to run on minicomputers. Also, as discussed below, all but one pass is machine invariant, reducing porting and maintenance overhead. Finally, almost no price is paid for this choice. The compiler spends most of its time in lexical analysis, opening files and reading source, or it spends its time in editing libraries or loading. The extra cost of invoking multiple passes is trivial. I/O time to read and write the files of intermediate language between passes could be significant, but careful design[18] of the intermediate language to pack it efficiently has prevented this from being a bottleneck.

The style of intermediate language is perhaps more controversial. We flatten the parse tree with respect to declarations and control structure, but we explicitly keep expressions in tree form. (Expression trees are linearized in the intermediate files between passes, but they are rebuilt before use.) Other compilers usually flatten expression trees too, representing them with triples or quadruples, or with instructions for some simple stack machine as in P-code. We prefer explicit trees because we want to transform expressions by inserting nodes, deleting nodes, reordering nodes, etc. We also want to perform operations that require walking the tree several times, not always in the same order. But perhaps more important is that flattening a tree loses information which is hard to recover. For instance, quadruples introduce large numbers of extraneous temporaries, and considerable effort is then required to collapse the redundant ones[10]. By contrast, our register allocation algorithm rarely introduces redundant temporaries. Other compiler writers have also noted how hard it is to make sensible use of a rich instruction set or multiple registers if the level of the intermediate language is too low[11,12,20]. For example, any indication of why some sequence of operations is being performed is completely lost, whereas if it were still available, it might be apparent that some addressing mode or some special instruction would enable the original computation to be done in a completely different way. This flaw is present in P-code: the problem is not that it is code for a stack machine, but that being code for such a simple stack machine induces overspecification of code sequences which requires considerable effort to optimize out again.

Having discussed the unusual aspects of the Zed compiler, let us return to its general organization. The phases into which it is divided are: *Lex* (lexical analysis), *Parse* (LALR(1) parsing), *Clean* (machine invariant code optimizations), *Data* (external data module creation and initialization) and *Code* (code generation). The first four phases are completely machine invariant except insofar as they may require the values of target machine environment parameters, such as *BYTES_PER_WORD*, which are contained in a file available to each phase. The library format and the relocatable object code format required by the loader are also completely machine invariant. Consequently, the only machine specific source appears in the single phase *Code*, and this is what must be rewritten to produce a compiler for a new machine.

Code is basically an automaton, driven by a machine invariant intermediate language produced by *Clean*, and it produces load code in the machine invariant format required by the loader. The automaton itself is machine invariant, as are many of the action routines it calls. Machine invariant routines are used, for example, to enter symbols in the symbol table, to rebuild expression trees from the input stream, to determine the type for each node of an expression, to record the start of a select (multiple alternative) statement, or to record the start of each successive alternative case within such a statement. In fact, other than a few simple actions such as emitting the code to enable or disable interrupts, the only actions that require machine specific implementations are those concerned with stack manipulation, with expression evaluation, and with control flow. A number of abstract models are used to facilitate the implementation of these action routines. Many of these models actually have machine invariant implementations, while the others at least have machine invariant interfaces.

Emitting Code

The first requirement in generating load code is the ability to output instructions or data. For each function that is being compiled, there are three address spaces which must be built, and which may remain disjoint, or at least will be relocated separately. These are the literal string space, the constant and table space, and the instruction space. Copying literal strings to the string space and recording appropriate addressing and relocation information in the symbol table is done by a machine invariant routine. Putting constants and tables into that address space can readily be done with service routines that implement a fairly low level of abstraction: a function, *Set_loc*, is called to change the address space to which items are being output, and two other functions, *Load_word* and *Rload_word*, are called to output data items whose values are absolute and data items whose values must be modified by relocation, respectively. The offset within the current address space at which the next item is to be loaded is available from an external variable, *Counter*. This abstraction serves to hide all the details of how load code is actually formatted and written, and the implementation is machine invariant.

For building code into the instruction space, however, we want a higher level abstraction. The actual construction of code for real machines can be quite complex. Instruction operands may require, or may be able to exploit, elaborate addressing modes. Constant operands may best be used as immediate operands, or may be collected in one of several kinds of pools. Jump instructions can take different forms, depending on the distance to the jump target. Forward references to items not yet defined (such as the targets of forward jumps) must be deferred and backplugged when the definition becomes available. If short address forms have been used, it is necessary to check, before each instruction is output, whether addressability to forward referenced items is about to be lost, and if so, to output the constant pool or intermediate jump target and emit a jump around this insertion. The compiler writer would prefer not to be concerned with these kinds of issues every time he produces a code burst. Instead, he would like to produce a sequence of instructions simply by a sequence of function calls that resemble assembly code.

This level of abstraction is provided by four functions: *Emit, Hop_gen, Jump_gen,* and *Backplug_gen.* The most basic of these is *Emit,* whose first operand indicates the machine instruction to be generated, and whose remaining operands are symbol table entries (or pseudo symbol table entries) for the operands of the machine instruction. The symbol table entries carry all addressing mode and relocation information as to how the operands might be addressed. Such uniform addressing implies having pseudo symbol table entries for operands that would not normally be in the symbol table, such as the registers or indirect addressing modes which use registers or memory words. (These are pseudo entries because their lifetime does not correspond to that of a normal symbol table entry - it may be as short as a single expression or longer than one function). *Emit,* however, produces exactly the specified instruction, and requires that all operands be fully defined when it is called. As stated above, this certainly is not appropriate for forward jumps, and may not be for some other jumps. Instead, the function *Hop_gen* is called with a jump condition (usually specified as one of the available jump instructions) and a label. If a backward jump is required it calls *Emit* to produce the appropriate instructions; for a forward jump it reserves space in the instruction segment for the shortest form of the appropriate jump and defers the generation of the actual instruction until the label's value becomes known. *Backplug_gen* is called, with a symbol as operand, at the point where the symbol is to appear. This triggers generation of any deferred instructions which are awaiting definition of the symbol. The last function, *Jump_gen,* is used to force an unconditional jump of a form that can go to any location in the address space, thus avoiding the jumps to jumps which might result from *Hop_gen's* forward jump strategy when short addressing is used.

Backplug_gen is machine invariant. *Emit, Hop_gen,* and *Jump_gen* are clearly machine specific. However, they can be implemented simply in terms of the lower level abstraction, together with a few service routines: *Check_addressability, Defer_gen, Backplug_symbol,* and *Cond_addr_gen.* Only the last two are machine specific: *Backplug_symbol* calls *Emit* to output deferred instructions, and *Cond_addr_gen* enters constants into constant pools and supplies addresses or immediate operands as required by the instruction type and symbol table description.

Stack Manipulation

Zed, like other languages in the BCPL family, is a stack based language in which users are encouraged to program using many small functions. This means that the efficiency of function calls, both in time and space, is crucial. Since Zed programs often have many small processes in the same address space, it is essential to check for stack overflow on function calls. Arguments are passed by value only, and the semantics are that a formal argument to a function is simply a local variable which may have been initialized in the call - "may have been" because any specific call can omit irrelevant trailing arguments. The called function can discover the number of arguments actually supplied by calling by a built-in function. Zed also provides a mechanism to allow functions that can be called with a truly arbitrary number of actual arguments, in which case the actual arguments supplied are obtained by invoking another built-in function. The design of the stack format and the call/return sequence must support these objectives.

This design, including the decision of what to include in a shared prolog and epilog (if any), is one of the most important stages in implementing Zed. On a typical machine there are six or more plausible designs to be considered and the one recommended by the manufacturer is rarely optimal. Some of the factors affecting the choice are: the exact form of the subroutine jump instruction, including implied registers or side effects (JSR on the DEC PDP-11, for instance, saves the return address in the specified register but also pushes the previous contents of that register onto the stack); any asymmetry in the forms of short addressing that should be used for high-use local variables (as on the IBM Series 1, the Modcomp IV, or the TI 990); the functionality provided or any asymmetry present in hardware supported stacks or stacking instructions (although it is rare for these to be of any use in implementing the stack frames required for high level languages); asymmetry in the direction of allowed offsets to base registers; the effect of a stack frame cache if provided; the cost of checking for stack overflow; and (of course) the execution times for different instructions. The style of Zed programming is to have many small functions, and many small processes. This means that, for most machines, keeping too much in registers is unwise because, statistically, more is lost by the required register saves and restores than is gained by the increased option of using register type instructions.

One somewhat unusual stack design, which has turned out to be very good for a number of machines, keeps the stack frame size fixed throughout the lifetime of a function, and uses a single stack frame base pointer which points to the top of the stack frame rather than to the usual bottom position adjacent to the caller's stack frame. With this design, checking for stack overflow is cheap, in that the stack frame base pointer need only be compared against its limit for this process. Although this check is not complete (because arguments to called functions are placed beyond this pointer), stack overflow will be caught when function entry to the called function is attempted. Use of this stack format does, however, necessitate expression modification to avoid function calls when collecting arguments to another function.

An abstraction, provided by several machine specific functions with machine invariant interfaces, allows these decisions to be encapsulated so as not to affect other parts of the compiler. Even such questions as whether the stack grows upward or downward are hidden. The functions *Prolog_gen* and *Return_gen* call *Emit* to generate function entry and exit. The three major parts of a stack frame, namely the space for local variables, the space for local vectors, and the space for temporaries used in expression evaluation, are each allocated as separate address spaces, and commands to the loader to relocate them into a single contiguous space are given by *Epilog_gen* after each function has been compiled. The functions *Next_auto, Next_autovec* and *Next_temp* allocate space in their respective segments.

Expression evaluation

Code receives expressions as trees in the intermediate language, rather than, for instance, as operations on a simple stack machine. The reason for this is that many machines have powerful instructions that are equivalent to large subtrees of expression trees, and if we want to exploit such instructions, it is much easier to recognize where they would be useful from the explicit tree representation rather than from peephole optimization on some already flattened representation of a tree.

The semantics of Zed specify a precise mapping of source language expressions onto binary trees and require evaluation by a tree walk in a strict left-right-root order: the compiler writer may deviate from this only where he can prove that bitwise identical results will be obtained, including all side effects. This requirement originates from the demand that compiled code have the same effect on all machines, and it is heavily relied upon by programmers. (Measurements on many thousands of lines of existing code show very little could be gained by a more liberal order of evaluation). A consequence of having a well defined order of evaluation is that most of the tree modifications permitted for optimization are machine invariant and will already have been done by the *Clean* phase. Machine specific modifications to the expression tree are performed by the function *Modify*. One of these modifications is to set the test location used for select statements, and another is to set the return location for function values. *Modify* may also invoke *Modify_calls*, which is a machine invariant function used with fixed size stack frame designs to rearrange the tree to avoid function calls during the loading of arguments for other function calls, as mentioned earlier. *Modify* invokes the machine specific function *Modify_opers* which walks the tree replacing any nodes or subtrees for which the particular machine has better operators.

Once the final expression tree is determined, each internal node is labelled with the location at which its result will be deposited, and the names of scratch registers used in its evaluation. The machine specific function *Allot_regs* does this by a tree walk. This walk is done in the order right-left-root, rather than in the more usual order left-right-root, so that the locations changed between the time an intermediate result is computed and when it will be needed are known when deciding where to save it. This enables *Allot_regs* to avoid storing an intermediate result in such a location. While this does not necessarily lead to optimal register allocation, in practice it is very good, and does avoid the problem with the more conventional allocation strategies that results left in registers may be moved more than once before they are used. Since real machines often restrict the choice of source, destination, and/or scratch registers for various instructions, we have developed an abstract model of register allocation. For each operator in an expression tree, we specify the register class required for its operands, and the register class required for its result. This specification is obviously machine specific, and indeed may depend on information about the operand subtrees. We define the locations where the operands are available to be the external operands, and if these locations do not satisfy the register constraints, we pick registers that do satisfy them and specify that the operands, after both have been evaluated, be loaded into these registers which we call the internal operands. The internal result, the location where the result will be produced, is often fixed once the internal operands are chosen. However, sometimes it is not, in which case it must be chosen subject to whatever register constraints apply. Finally we choose the external result, that is, some safe location where the value computed at the node can be left until it is needed. All choices are made (using bit vectors defining the available registers) in such a way as to minimize unnecessary moves and to minimize the number of registers and temporaries used. Although the functions which implement all this are machine specific, this organization appears to be readily adapted to any machine we have encountered so far, and the register and temporary allocations it makes are in practice very good.

Once the allocation of registers and temporaries is complete, it would be simple to have the function *Arith_expr_gen* walk the tree, checking various conditions and calling *Emit* to generate the best instructions to evaluate the nodes into the result locations indicated. Such a procedural approach, however, often produces a bulky compiler if we try to take advantage of all the special cases for which the instruction set provides. The usual solution to this would be to have *Arith_expr_gen* use code tables, made up of templates which match possible subtrees, and code bursts to be produced if a template is chosen. Unfortunately, for a machine with a complex structure, these code tables can also be bulky because of redundancy. We achieve a balance between these approaches by another abstract model we have developed, that of a code generating interpreter. This machine walks an expression tree, driven by a table that, for each operator which may label a node, specifies possible conditions that might exist at that node and the sequence of actions to be taken if the conditions are met. The actions normally involve calls to *Emit, Hop_gen,* and *Backplug_gen* with the internal operands of the node, its scratch registers and its external result - as for a conventional table driven code generator. The possibility of other actions, especially calling functions or reentering the table to expand macros, justifies viewing it as a more powerful machine. Again the table, and the functions which define the conditions, expand the operands and take the actions, are all machine specific, yet implementing this model for a new machine seems to be easy, and the resulting compiler is small even when it fully exploits a rich and irregular instruction set.

Control flow

Although Zed has powerful modern control structures, the earlier phases of the compiler have reduced these so that *Code* is only faced with three control types: unconditional jumps, jumps depending on the truth value of an expression which must be evaluated, and the select (multiple alternative) statement. Unconditional jumps are trivial in that they may be done by a machine invariant call to *Jump_gen.*

Conditional jumps are more interesting. Through classical (and machine invariant) techniques, the logical AND, OR, and NOT functions are expanded so that we are only interested in expression trees for which the top node is a simple variable, an arithmetic operator or a relational one. Registers and temporaries have been assigned just as for any expression tree. *Relation_gen,* the machine specific function which produces the conditional jump, knows how to test a simple variable for zero (FALSE) or nonzero (TRUE), on the specific target machine. Many machines have some kind of condition code which is set by arithmetic or comparison instructions. On such machines, for expression trees with arithmetic operators as the top node, *Relation_gen* uses *Arith_expr_gen* to generate code to evaluate the expression tree and then makes use of its knowledge of whether the specific operator at the top node set the condition code usefully in order to know whether a specific test for zero or nonzero is needed before generating a jump on condition code. Again on such machines, for expression trees with a relational operator as the top node, *Relation_gen* calls *Arith_expr_gen* to generate code to evaluate each of the operand subtrees of a relational operator, then calls a machine specific function *Set_condition_register* to produce the comparison instructions so that *Relation_gen* can generate the appropriate jump on condition codes. There is a second class of machines which can perform conditional transfers as a side effect of the same instruction that does the comparison or arithmetic operation. For such machines, *Relation_gen* can call *Arith_expr_gen* to generate code to evaluate the operand subtrees, but it must issue the instructions to evaluate the top node itself. Although *Relation_gen* is machine specific, the appropriate version for any particular machine can readily be transcribed from one of only a few prototypes.

There are two forms of the select statement in Zed. One form, the string select, matches a test string against a set of literal strings, and then executes the action corresponding to the matching string, if any. The other form, the word select, matches a test value against a set of constant values and intervals, and then executes the action corresponding to the match, if any. Machine invariant routines in the *Code* phase record the beginning of a select statement, arrange that the test value is put in a test cell, and generate a jump to the multiway branching code to be produced at the end of the set of alternative cases. As each case is compiled, the condition that selects it and the starting address of the code for it are recorded. At the end of the select statement, machine specific routines are called to generate the multiway branch. *Strselect_gen*, called for string selects, often implements the test as a loop that calls the standard library function *.Equal* to compare strings for identity, breaking out of the loop when a match is found. On some machines, however, it is cheaper simply to generate inline code to compare the strings.

For word selects, the compiler chooses amongst four different implementation strategies. *Straighttest_gen* is called to generate code equivalent to a sequence of if statements when there are so few cases that this produces the most compact code. When more than half the possible values between the smallest and the largest cases have corresponding actions, *Jumptable_gen* is called to generate code to check that the test value is in the appropriate interval and, if so, to jump indirect through a jump table. For select statements not implemented by the above, if there are any interval cases *Pairtest_gen* is called to generate code to do the matching. All cases are represented as intervals (although some may consist of a single element), the intervals are sorted, and code is produced to search sequentially to find the interval enclosing the test value. Finally, for the remaining select statements, *Binarysearch_gen* orders the cases in a heap-like structure without pointers so that the matching case can readily be found by a binary search, and produces the appropriate code.

All five of these functions are easy to write for a new machine, by following the existing functions for other machines as models, and using *Emit*, *Hop_gen*, *Jump_gen*, and *Backplug_gen* to produce the machine instructions and *Set_loc*, *Load_word*, and *Rload_word* to build the necessary tables.

Conclusions

The Zed compiler originated in the spring of 1976 as a compiler for the typeless language Eh. It has successfully adapted to changes in the language as it evolved, including the introduction of types (which led to the name change) in the summer of 1978. The compiler has so far been ported to half a dozen very different machines, and the critical issues in porting it to another half dozen machines have been examined in detail. The structure has proven general enough to cope with all of them.

Our principal conclusion is that the structure of the Zed compiler has been successful in enabling us to port the compiler easily to new machines. The time for one person to perform such a port has varied from one month to six. (This compares very favourably with the time required for porting other portable compilers, and even compares favourably with the time required to write an interpreter for Pascal-P[2,4,11,13,15]). Very little of this time is spent coding the machine specific functions. Instead, time is spent where it should be: on the design of the stack, analysis of register constraints, and determination of code bursts so as to take the best possible advantage of the available instruction set. The resulting code is statistically almost as good as an assembly language programmer can do while satisfying the language semantics.

Another effect of the chosen structure is that the resulting compilers are remarkably free of bugs. For example, in one implementation for a machine we had never used before, after the first clean compilation of the machine specific functions only a total of 92 bugs were found in a year of heavy use. Of these, 67 were found within the first 13 days of testing. Many of the remaining bugs were misunderstandings of how the machine hardware worked, or could be classified as performance bugs: the code produced worked correctly but was not as good as possible. All these bugs were easily corrected.

Acknowledgements

In addition to the authors, several other members of the portability group have contributed to this compiler. The first versions of it were written by Reinaldo Braga. The *Lex*, *Parse*, and *Clean* phases were written by Gary Stafford. The loader was written by Gary Sager. Fred Young did two of the ports.

References

1. Basili, V.R., and Turner, A.J., "A Transportable Extendable Compiler", Software-Practice and Experience, Vol 5 No 3 (July-September 1975), pp 269-278.

2. Berry, R.E., "Experience with the Pascal P-Compiler", Software-Practice and Experience, Vol 8 No 5 (September-October 1978), pp 617-627.

3. Bonkowski, G.B., "The Structure of the Eh Compiler: Code Generation", Computer Science Department, University of Waterloo, August 1978.

4. Bron, C., and De Vries, W., "A PASCAL Compiler for PDP 11 Minicomputers", Software-Practice and Experience, Vol 6 No 1 (January-March 1976), pp 109-116.

5. Cattell, R.G., "A Survey of Some Models of Code Generation", Department of Computer Science, Carnegie-Mellon University, November 1977.

6. Cattell, R.G., "Formalization and Automatic Derivation of Code Generators" Technical Report CMU-CS-78-115, Department of Computer Science, Carnegie-Mellon University, April 1978.

7. Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R., "Thoth, a Portable Real-Time Operating System", CACM, Vol 22 No 2 (February 1979), pp 105-115.

8. Colin, A.J.T., Shorey, K., and Teasdale, W., "The Translation and Interpretation of STAB-11", Software-Practice and Experience, Vol 5 No 2 (April-June 1975), pp 123-138.

9. Glanville, R.S., "A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers", Technical Report UCB-CS-78-01, Computer Science Department, University of California, Berkeley, December 1977.

10. Gries, D., "Compiler Construction for Digital Computers", John Wiley and Sons, 1971.

11. Grosse-Lindemann, C.O., and Nagel, H.H., "Postlude to a PASCAL-Compiler Bootstrap on a DECSystem-10", Software-Practice and Experience, Vol 6 No 1 (January-March 1976), pp 29-42.

12. Haddon,B.K., and Waite, W.M., "Experience with the Universal Intermediate Language Janus", Software-Practice and Experience, Vol 8 No 5 (September- October 1978), pp 601-616.

13. Lecarme, O., and Peyrolle-Thomas, M.-C., "Self-Compiling Compilers: An Appraisal of their Implementation and Portability", Software-Practice and Experience, Vol 8 No 2 (March-April 1978), pp 149-170.

14. Newcomer, J.M., "Machine Independent Generation of Optimal Local Code", Department of Computer Science, Carnegie-Mellon University, 1975.

15. Neal, D., and Wallentine, V., "Experiences with the Portability of Concurrent PASCAL", Software-Practice and Experience, Vol 8 No 3 (May- June 1978), pp 341-353.

16. Richards, M., "The Portability of the BCPL Compiler", Software-Practice and Experience, Vol 1 No 2 (April-June 1971), pp 135-146.

17. Sager, G.R., "The Thoth Linking Loader", Technical Report CS-77-15, Computer Science Department, University of Waterloo, October 1977.

18. Stafford, G.J., "Structure of the Eh Compiler: Lexical Scanning, Syntactic Analysis, and Optimization", Computer Science Department, University of Waterloo, December 1978.

19. Tanenbaum, A.S., "Implications of Structured Programming for Machine Architecture", CACM, Vol 21 No 3 (March 1978), pp 237-246.

20. Waite, W.M., "Intermediate Languages: Current Status", Workshop on Portability of Numerical Software, Oak Brook, Ill., (June 1976).

21. Wulf, W.A., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C., "The Design of an Optimizing Compiler", American Elsevier, 1975.