Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Marc Brandis
Régis Crelier
Michael Franz
Josef Templ

The Oberon System Family

April 1992

174

Authors' addresses:

Institut für Computersysteme
ETH Zentrum, CH-8092 Zurich, Switzerland
e-mail: brandis/crelier/franz/templ@inf.ethz.ch

The Oberon System Family

M. Brandis, R. Crelier, M. Franz, J. Templ
Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

SUMMARY

Oberon simultaneously refers to a modular, extensible operating system and an object-oriented programming language developed for its implementation. While the original Oberon System had been conceived as the native operating system for a custom-built workstation, further implementations for several commercial platforms were developed later and are described here. All of these implementations are based on an efficient, retargetable Oberon compiler, and each provides a complete Oberon environment and the original library interface. This paper describes the structure of the compiler, summarizes the experience gained in adapting it for various CISC and RISC processors, and presents some empirical performance data. It also sheds light on the task of grafting an operating environment onto an existing operating system.

KEY WORDS Operating System Compiler Code Generation Software Portability Oberon

INTRODUCTION

Late in 1985, J. Gutknecht and N. Wirth initiated the *Oberon Project* with the goal of developing a complete operating system for small workstations "from scratch". Their implementation was targeted towards the personal workstation *Ceres* [1] and first presented in 1988. As a by-product of the operating system design effort, a new programming language and a compiler for it came into being. Most of the operating system, as well as the compiler itself, are written in this new programming language. The new operating system and programming language were both given the name of the project, and are now called the *Oberon System* and the *Oberon Language*, respectively [2, 3].

Encouraged by the attention that the Oberon language and system received in the computing community, a second project was started in early 1989, with the goal of making Oberon available to interested parties without access to Ceres computers. At that point, Ceres computers were unavailable outside of ETH. The first person engaged in this second project became R. Crelier, who developed a portable compiler for the Oberon language [4], in which the machine-independent parts were separated from the target-machine specific details. It was recognized that the effort of porting a compiler to other target architectures could be reduced significantly by encapsulating the machine-independent parts, so that the authors of new code generators need not understand the parser and symbol table handler. Like the original Oberon compiler, from which it was derived, the portable compiler was itself written in the Oberon language and generated code for the Ceres, which is based on a National Semiconductor 32000 processor [5].

M. Franz joined the project in mid-1989. His task was to create a second code generator for the portable Oberon compiler that would output object code for Motorola 68020 processors [6]. The envisaged target machine was an Apple Macintosh II computer [7]. Soon afterwards, J. Templ became another member of the team, his goal being a code generator for the Sun SPARC architecture [8, 9]. Meanwhile, R. Crelier kept on refining his portable compiler, improving the front-end, streamlining the interface to the code-generator, and documenting it. By the end of 1989, first versions of the code generators for Motorola 68020 and SPARC were operational.

At this point it was decided that all of the Oberon system should be ported, and not only the compiler. This would be aided greatly by the fact that the Oberon system was itself written in Oberon, with minor exceptions. In the months that followed, M. Franz and J. Templ implemented the low-level functions of the Oberon system for their respective target machines, while R. Crelier, whose portable compiler had reached a stable state by then, started on the development of a fourth code generator, for the MIPS R2000 processor architecture [10]. By mid-1990, first versions of Oberon for Macintosh II and SPARC were running, and in early 1991 a prototype for the Digital Equipment DECstation was completed.

Around the same time, a fourth person, M. Brandis, entered the project with the intent to develop a code generator for the IBM RISC System/6000 architecture [11, 12]. Later on, he was able to port the rest of the system to the target machine quickly, because portable interfaces to the UNIX file system [13] and the X Window System display library [14] as well as a portable memory allocator and garbage collector existed already from other Oberon implementations. Oberon for the RS/6000 was released in mid-1991. At the same time, the portable Oberon compiler was modified slightly to accept Oberon-2 programs [15].

From September to December 1991, a new workstation later christened Chameleon [16] was developed around the LSI Logic LR33000 MIPS Embedded Processor [17], which is compatible to the MIPS R3000. The Oberon System was ported as the native operating system to this machine in January 1992 by R. Crelier. This took only one month because the existing code generator for the DECstation could be used.

Today, Oberon is available on five different processor architectures, and further versions are still forthcoming. Moreover, several companies have acquired the source code of our portable Oberon compiler and are developing, or planning to develop, commercial implementations of Oberon.

ARCHITECTURE OF THE COMPILER

In contrast to other programs written in a high-level programming language, a compiler has to be modified to produce code for a new machine. Therefore, it is worthwhile paying attention to portability before writing it. The time invested in designing a well structured compiler, separating machine-independent from machine-dependent parts, is rewarded many times when porting it. However, many compilers developed with the goal of being portable have turned out to be not only portable, but inefficient in terms of compilation speed and quality of compiled code. Portability and efficiency have been given equal importance in our approach. Hence, automated retargetable code generation has not been considered. Instead, we looked at more conventional and faster techniques, in particular single-pass compilation.

In a single-pass recursive-descent compiler, all phases of compilation are executed "simultaneously", i.e. actions of syntax analysis, code generation, type checking etc, are interleaved. Because all required attributes are passed on the procedure stack, no intermediate representation of the source text is needed between the different phases. This makes the compiler compact and efficient, but not easily portable. Indeed, since machine-dependent and machine-independent phases are closely coupled, it is difficult to modify the compiler for a new machine. One solution to the problem is to clearly separate the compilation phases into two groups: a *front-end* consisting of the machine-independent phases (lexical and syntactic analysis, type checking) and a *back-end* consisting of the machine-dependent phases (storage allocation, code generation).

Effectively, compilation thereby becomes a two-pass process, although the source text is processed only once. The interface between front-end and back-end is a complex data structure in memory instead of a sequential file, taking advantage of large stores. Only the back-end needs to be modified when the compiler is ported. The front-end enters declarations into a symbol table and builds an abstract syntax tree representing the program statements. If no errors are found, control is passed to the back-end, which generates code on the basis of this syntax tree. Since this structure is guaranteed to be free of errors, type checking or error recovery are not part of the back-end, which is a noteworthy advantage. Only implementation restrictions must be checked for. Another advantage of an intermediate representation is that additional passes may be inserted to improve code quality. Such an optimization phase cannot be embedded easily in a conventional single-pass compiler, if at all.

Module Structure

The front-end and the back-end are implemented separately as a set of nine modules, all written in Oberon.

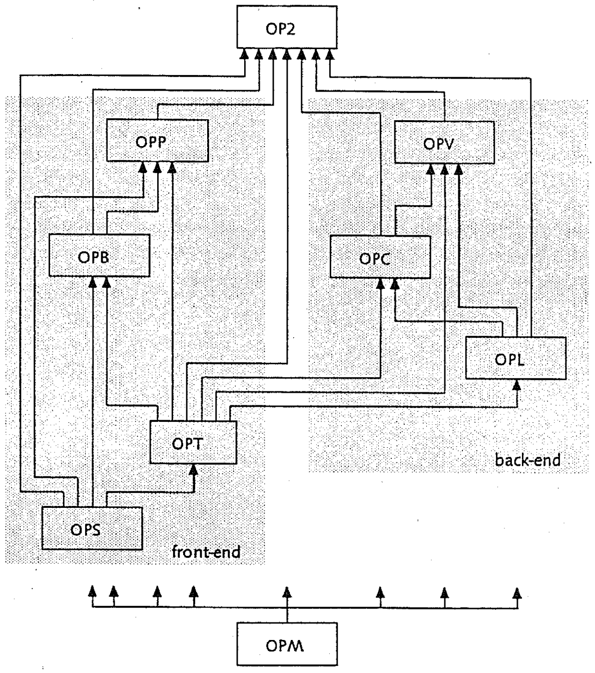


Figure 1 Module import graph (an arrow from A to B means B imports A).

The lowest module of this hierarchy is OP*M*, where *M* stands for *m*achine. We must distinguish between the host machine on which the compiler is running, and the target machine for which the compiler is generating code. Most of the time, the two machines are the same, except when using a cross-compiler. OPM defines and exports several constants used to parametrize the front-end. Some of these constants reflect target machine characteristics or implementation restrictions. For example, these values are used in the front-end to detect overflow conditions in the evaluation of constant expressions. But OPM has a second function, too. It works as the interface between the compiler and the host machine. This interface includes procedures to read the text to be compiled, to read and write data in symbol files [18], and to display text (e.g. error messages) onto the screen. All these input and output operations are strongly dependent on the operating system. The compiler is structured in such a way that it can be easily ported to environments other than the Oberon System. If the compiler resides in the Oberon System environment, the host-dependent part of OPM is based on the standard modules Texts and Files.

The topmost module OP2 is very short. It is the interface to the user, and therefore host machine-dependent. Like the host-dependent part of OPM, this module remains unchanged when the compiler is used in the Oberon System environment. It first calls the front-end with the source text to be compiled as a parameter. If no error is detected, it then calls the back-end, passing the syntax tree that was generated by the front-end.

Between the highest and the lowest module, one finds the front-end and the back-end, which consist of four and three modules, respectively. During compilation, there is no interaction between these two sets of modules. The symbol table and the syntax tree are defined in module OPT and are accessed both by the front-end and the back-end. This explains the presence of import arrows from OPT to back-end modules visible in the import graph above. But there is no transfer of control, such as procedure calls.

The front-end is controlled by module OPP, a recursive-descent parser. Its main task is to check syntax and to call procedures constructing the symbol table and the syntax tree. The parser requests lexical symbols from the scanner (OPS) and calls procedures of OPT, the symbol table handler, and of OPB, the syntax tree builder. OPB also checks for type compatibility.

The back-end is controlled by OPV, the tree traverser. It first augments the symbol table with machine-dependent data (using OPM constants), such as the size of types, the address of variables, or the offset of record fields. It then traverses the syntax tree and calls procedures of OPC, the code generator, which in turn synthesizes machine instructions using procedures of OPL, the low-level code emitter.

This module structure results in a fully portable front-end, as well as a host-machine independent back-end.

Symbol Table

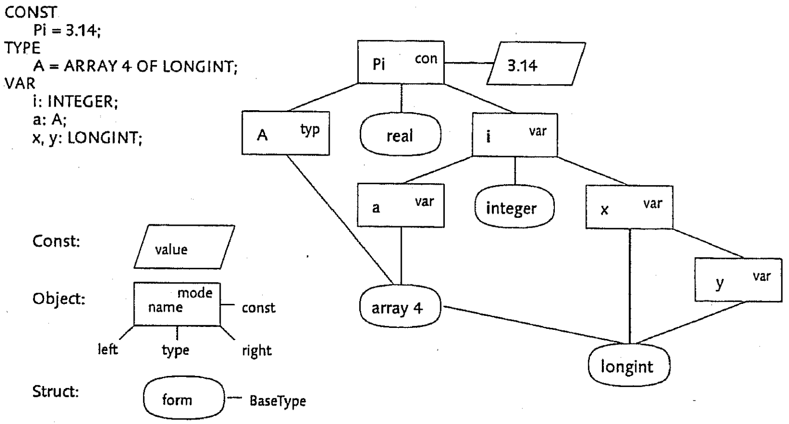
The symbol table contains information about declared constants, variables, types, and procedures. It is built by the front-end. The front-end uses it to check the context conditions of the language and the back-end retrieves type information from it. The symbol table is a dynamically allocated data structure with three different component types: *Object*, *Struct*, and *Const*.

An *Object* is a record (more precisely a pointer to a record), which represents a declared, named object like a constant, a type, a variable, or a procedure. The name of the object, stored in the object itself, is used as a key to retrieve the object in its scope. Each scope is organized as an ordered binary tree of objects and is anchored in the owner procedure, which in turn belongs as an object to the enclosing scope. Parameters of the same procedure, fields of the same record, and variables of the same scope are additionally linked together sequentially in order to maintain the declaration order. Procedures which do not call any further procedures (leaf procedures) are marked by the front-end, as are variables whose addresses are never needed, and which therefore can be allocated in registers. The back-end may use this information for improving code quality. Note that this information would not be available in a single-pass compiler without an intermediate representation of the program.

An object always has a type, described by a *Struct* record pointed to by a field in the object. There are several classes of types: basic types such as character, integer, or set, and composite types like array, open array, or record.

The third element type of the symbol table is *Const*. This record contains numeric attributes of objects, such as values of declared or anonymous constants.

An example of a symbol table is shown below.



```
CONST
   Pi = 3.14;
TYPE
   A = ARRAY 4 OF LONGINT;
VAR
   i: INTEGER;
   a: A;
   x, y: LONGINT;
```

Figure 2   Declarations and corresponding symbol table.

Syntax Tree

The front-end builds an abstract syntax tree representing all statements of the program. The Oberon syntax is mapped onto a tree of elements called *Nodes*. Each Oberon construct can be decomposed into a root element identifying the construct and a maximum of two subtrees representing its components: an assignment has a left and a right side, a While statement has a condition and a sequence of statements, and so on. Some Oberon constructs are organized sequentially, e.g. lists of actual parameters in procedure calls and sequences of statements in structured statements. Auxiliary nodes might have been inserted to link these subtrees, yet an additional link field in the node is more space-efficient.

Each node has a class, and possibly a subclass, identifying the Oberon construct represented. It also has a type, which is a pointer to a *Struct* of the symbol table. Similarly, a leaf node representing a declared object contains a pointer to the corresponding *Object* of the symbol table. A *Const* may be attached to a node to describe a numeric attribute, such as the value of an anonymous constant.

The position in the source text is stored in the root node of each statement. This facilitates locating compilation errors reported by the back-end. Figure 3 shows the representation of two statements manipulating variables declared in Figure 2.

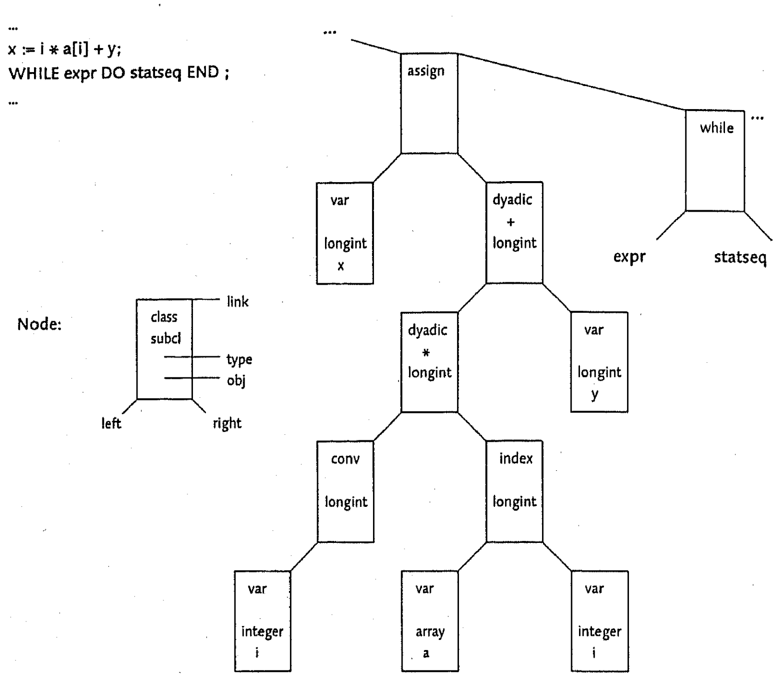


Figure 3 Statements and corresponding syntax tree.

The intermediate representation might have been a stream of instructions for a virtual machine, but we preferred to use an abstract syntax tree for various reasons. A virtual machine instruction set would have been defined without any knowledge of future target machines. Perhaps the mapping of this instruction set to a real instruction set would not be easy, the virtual and real machines being very different (RISC vs. CISC e.g.). Furthermore, generating these pseudo-instructions requires a code generator already, whereas building the syntax tree is a trivial recursive task easily embedded in a recursive-descent parser.

Since the tree is a natural mapping of the Oberon syntax, each procedure of the parser returns as parameter the root of the subtree corresponding to the construct just parsed. Furthermore, a tree keeps the program structure intact, so that control-flow dependent optimizations can be integrated easily. Without a tree, an expensive control-flow analysis would be required, since basic blocks would have been dissolved in the linear code. The reordering of program pieces is easier to perform in a tree than in an instruction stream. For example, by first generating the statement sequence of a While statement (right subtree) and then evaluating the condition (left subtree), one branch can be removed from the loop.

RETARGETING THE COMPILER

Writing a new compiler back-end for a particular target architecture is certainly the main effort in porting Oberon to another machine. Although all back-ends have a similar structure and deal with similar problems, the impact of a target processor's architecture is visible throughout the back-end. For us, it was an interesting experience to investigate the impact of RISC architectures in regard to the complexity of code generation.

The input to the back-end are two dynamically allocated data structures, namely the symbol table and the syntax tree. The output of the back-end is the decorated symbol table and the object code stored as a linear array of machine instructions.

Decorating the Symbol Table

The first task of the back-end is to add target-machine specific attributes to the objects of the symbol table. These attributes depend on the kind of each object as shown in Table 1.

Kind of Object	Added Attributes
type	size, type descriptor address
variable	address or register number
record field	offset
procedure	frame size
exported ~	entry number
method ~	method number

Table 1   Target specific attributes.

In the following, we will discuss the attributes mentioned in Table 1.

For type objects the size has to be calculated. As Oberon programs may perform run-time type tests, a type descriptor object has to be allocated and its address (or some kind of reference number) has to be associated with the type object.

Variable objects may either exist in memory or in registers. In the case of small register sets (NS32000, MC68020), all variables reside in memory. Otherwise (SPARC, MIPS, RS/6000) variables may reside either in memory or in registers. We decided to use a simple register allocation strategy based on the textual occurrence of a variable. The first *n* unstructured local variables (including the first *p* parameters) of a procedure are kept within registers. See Table 2 below for the target specific constants *n* and *p*. Only variables which are never referenced by their memory address are allocated in registers. This information is provided by the front-end. Variables of a structured type (Records, Arrays) always reside in memory. It turned out that this simple strategy allocates almost all-time critical local variables in registers and leads to reasonably efficient code without extra optimizations. It is also possible for the system-level programmer to influence register allocation by changing the declaration order of variables and by avoiding referential access to time-critical variables (e.g. by not passing them as VAR parameters).

Record fields receive an offset specifying their position in the enclosing record object. For procedures, a frame size sufficient to keep all locally defined variables is calculated. Exported procedures additionally need an identifying number so that they can be referenced from a client module. Oberon-2 type-bound procedures (methods) get a number which is used as an index into a method table.

In order to provide for an interface of Oberon programs and data structures to the underlying operating system, it is preferable to adopt the conventions of the target machine, for register usage, parameter passing, stack organization, and data alignment. However, these conventions are not always designed for maximum run-time efficiency, as can be seen especially on the Macintosh (see [19] for a more detailed discussion).

Table 2 shows the chosen size and alignment of standard types in the various Oberon implementations where *x / y* means that variables of that type are represented in memory by *x* bytes and their addresses are aligned to a multiple of *y*. The last column shows the above mentioned constant *n* for the maximum number of variables (including *p* parameters) allocated within the general purpose register set.

Target	SHORTINT	INTEGER	LONGINT	SET	REAL	LONGREAL	$n(p)$
NS32000	1 / 1	2 / 2	4 / 4	4 / 4	4 / 4	8 / 4	0 (0)
MC68020	1 / 1	2 / 2	4 / 2	4 / 2	4 / 2	8 / 2	0 (0)
SPARC	1 / 1	2 / 2	4 / 4	4 / 4	4 / 4	8 / 8	14 (6)
MIPS	1 / 1	2 / 2	4 / 4	4 / 4	4 / 4	8 / 8	12 (4)
RS/6000	1 / 1	2 / 2	4 / 4	4 / 4	4 / 4	8 / 8	18 (8)

Table 2 Conventions used for Storage Allocation.

Code Generation Overview

The second task of the back-end is to transform the syntax tree into a linearized sequence of machine instructions. This process can be thought of as mapping every node of the syntax tree into a semantically equivalent code sequence, and storing these code pieces linearly into an array. Every code piece consists of zero or more machine instructions and, in general, depends on code generated for other nodes. These dependencies should be kept as small as possible, in order to allow for an efficient and systematic code generation process. A simple rule to enforce a high degree of locality in the code generator is to require that dependencies exist between adjacent nodes only. In this case we can model dependencies as attributes flowing along the edges of the syntax tree. Depending on the target machine's architecture, attribute flow varies significantly as shown in Figure 4.

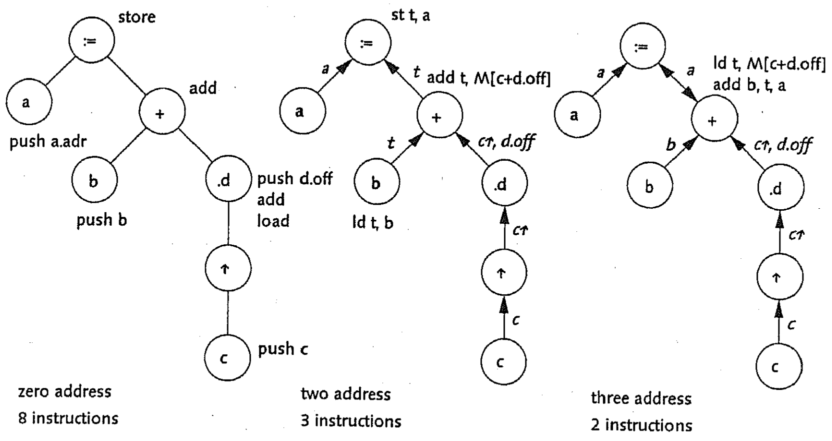


Figure 4 Syntax Tree and Attribute Flow for "a := b + c↑.d".

Given a zero-address machine (a stack-architecture), code can be generated for an operator node without knowing about its operand nodes, since all operands are pushed onto the stack. There are no attributes passed between nodes. This strictly context free schema cannot be preserved on non-stack-architectures. On two-address machines (such as the NS32000 and MC68020), on which operators have two explicit source operands, one of which is implicitly taken as the destination, the generated code depends on the locations of the source operands. Therefore, in this case source operand attributes (like memory address or register number) are passed from the operand to the operator nodes (bottom-up attributes). Three-address machines (like the SPARC, MIPS, or RS/6000) have instructions that evaluate two source operands into an explicitly specified destination. In this case additional attributes flow from operator nodes to operand nodes (top-down attributes). The number of operands directly corresponds to the complexity of the attribute flow and inversely corresponds to the number of generated machine instructions.

The code generator has to visit operand nodes before operator nodes (post-order) and destination operands before source operands (left to right) in order to generate attributes before they are used. The generated code pieces may be linearized by simply appending them into the linear code array as they are generated, i.e. a single tree traversal suffices for the code generation process. Since the attribute flow corresponds to the the call order in the recursive tree traversal, attributes can be passed as parameters on the stack instead of being stored in the syntax tree. The attribute records are commonly called *items*. For efficiency reasons all items are passed by reference. An item may be thought of as a variant record tagged with the operand's mode and additional mode-specific fields. Possible modes include those introduced by the Oberon language (Var, VarPar, Field, Proc, Const, ...) and those introduced by the addressing modes of the target machine (e.g. Register, Indirect, Absolute, Immediate, ...). For the purpose of demonstrating how the overall code generation works, we outline the involved procedures for a three-address machine and show a trace for compiling the simple statement of Figure 4. Procedure skeletons are presented in a top-down order starting with tree traversal on the statement sequence level.

```
PROCEDURE statseq(n: Node);
  VAR x, y: Item;
BEGIN
  WHILE n # NIL DO
    CASE n↑.class OF
      | Nassign: design(n↑.left, x); y := x; expr(n↑.right, y); Assign(x, y)
      | ...
    END ;
    n := n.next
  END
END statseq;

PROCEDURE design(n: Node; VAR z: Item);
BEGIN   (* no destination hint required *)
  CASE n↑.class OF
    | Nvar: IF InRegister(n.obj) THEN z.mode := Reg; z.adr := n.obj.adr ELSE ... END
    | Nfield: design(n↑.left, z); Field(z, n↑.obj↑.adr)
    | Nderef: design(n↑.left, z); DeRef(z)
    | ...
  END
END design;

PROCEDURE expr(n: Node; VAR z: Item);
  VAR x, y: Item;
BEGIN   (* z used as destination hint *)
  CASE n↑.class OF
    | Ndop:   (* dyadic operator *)
      x.mode := Undef; expr(n↑.left, x);
      y.mode := Undef; expr(n↑.right, y);
      CASE n↑.subcl OF
        | plus: Add(x, y, z)
        | ...
      END
    | ...
    ELSE design(n, z)
  END
END expr;
```

Figure 5   Syntax Tree Traversal.

These three procedures constitute the tree traversal process. In order to separate tree traversal and code generation clearly and to keep procedure sizes moderate, code generation is performed in separate routines sketched below. As the destination attributes can only be used in certain cases (e.g. on load/store architectures if and only if the destination is a register), the destination attributes are interpreted as hints. They are ignored if the destination operand cannot be used in the generated instruction (e.g. if mode = Undef). Therefore code generation for an assignment finishes with an explicit call of Assign which is supposed to generate code for an assignment to any destination operand. As a local optimization, Assign(x, y) does not generate code if *x* and *y* are the same register. Note that for evaluating expressions, the destination item can be propagated to sub-expressions for unary operators only. Operands of binary operators must, in general, be evaluated into a temporary variable because the destination operand may occur in other sub-expressions as well.

```
PROCEDURE Assign(VAR x, y: Item);
BEGIN
  IF (x.mode = Register) & (y.mode = Register) THEN    (* register move *)
    IF x.adr # y.adr THEN Emit(add, y, zero, x) END
  ELSE ...
  END
END Assign;

PROCEDURE Add(VAR x, y, z: Item);
BEGIN
  IF x.typ↑.form = Int THEN Emit(add, x, y, z)   (* integer addition *)
  ELSIF ...
  END
END Add;

PROCEDURE DeRef(VAR z: Item);
PROCEDURE Field(VAR z: Item; offset: LONGINT);
PROCEDURE Emit(op: INTEGER; VAR x, y, z: Item);
PROCEDURE GetReg(VAR x: Item);
```

Figure 6   Code Generation.

Emit is expected to generate code for a given three-address operator. On load/store architectures, the operands must be registers or small literals. Therefore Emit first synthesizes code to load the second operand (mode = register indirect) into a register and then code to perform the addition. Code generation for two-address machines is similar except that there is no hint parameter (see "Expression Evaluation" below).

The following trace for compiling the statement "a := b + c↑.d" represents procedure calls by indentation and value and reference parameter passing by assignments and equal signs respectively. It is assumed that variables *a*, *b*, and *c* reside within registers (which is the typical case for RISC machines).

```
stat("a := b + c↑.d")
  design("a", in: Undef, out: $Reg_a$)
  expr("b + c↑.d", in: $Reg_a$, out: $Reg_a$)
    expr("b", in: Undef, out: $Reg_b$)
      design("b", in: Undef, out: $Reg_b$)
    expr("c↑.d", in: Undef, out: M[$Reg_c$+$Off_d$])
      design("c↑.d", in: Undef, out: M[$Reg_c$+$Off_d$])
        design("c↑", in: Undef, out: M[$Reg_c$+0])
          design("c", in: Undef, out: $Reg_c$)
          DeRef(in: $Reg_c$, out: M[$Reg_c$+0])
        Field(in: M[$Reg_c$+0], out: M[$Reg_c$+$Off_d$], $Off_d$)
    Add($Reg_b$, M[$Reg_c$+$Off_d$], in: $Reg_a$, out: $Reg_a$)
      Emit(add, $Reg_b$, M[$Reg_c$+$Off_d$], in: $Reg_a$, out: $Reg_a$)
        reserve register t
        "ld t, [c + $off_d$]"
        "add b, t, a"
  Assign($Reg_a$, $Reg_a$)
```

Figure 7   Call Trace.

The outlined procedures are associated with different levels of abstraction. Those with Node-parameters are responsible for tree traversal, those with Item-parameters are responsible for code selection and the remaining operations are used for emitting machine instructions. We have a layered architecture of the code generator consisting of the *tree traversal* at the top, the *code selection* procedures in the middle, and the abstract data structure *code array* at the bottom. This architecture is reflected by a hierarchy of three modules, which is common to all of our Oberon compiler back-ends.

Module	Abstraction	Examples
OPV	tree traversal	statseq, design, expr
OPC	code selection	Assign, Add, Field, DeRef
OPL	code array	Emit, GetReg

Table 3 Layered Architecture of the Code Generator.

All of these Oberon compiler back-ends have been implemented as cross compilers running on a host computer (Ceres) but producing code for a particular target machine. This well-known technique leads to back-ends running on the target machine after a self-compilation step on the host machine. The result of the first self-compilation has to be the same as the result of further self-compilations on the target machine (fixpoint test). It this test fails, there is at least one remaining compiler error. Byte and bit ordering turned out to be the only problem in implementing portable cross compilers (and later on in implementing portable system-level programs). The encountered formats are shown in Table 4 below.

Target	byte ordering	hex value of {0}
NS32000	little endian	00000001
MC68020	big endian	00000001
SPARC	big endian	00000001
MIPS	little endian	00000001
RS/6000	big endian	80000000

Table 4 Byte and Bit Ordering.

The next chapters describe special aspects of the code generator in more detail. These include accessing global and local variables, procedure activation, handling of temporary values, procedure entry and exit code, expression evaluation, interlocks, run-time checks, type bound procedures, and target specific optimizations. Some measurements conclude the chapter on back-ends.

Global Variables

In the Oberon programming language, one can distinguish between global and local variables. Global variables are declared outside of any procedure and may be accessed as long as the enclosing module is present in the system. The obvious way to allocate memory for them is to assign to each global variable in a module a fixed position in a memory block, which is allocated at the time the module is loaded into the system. This is the method used in all implementations of Oberon.

One distinction between different systems is whether they keep a special pointer to this global memory block or not. If there is a register reserved for a pointer to the global data area, a global variable can be accessed by addressing it relative to this pointer. This generally results in short addresses, but also has the disadvantage that a new pointer has to be loaded into the respective register whenever a procedure in another module is called. The alternative is to address all global variables using their absolute addresses, which have to be inserted by the linker at load time. That way, no pointers have to be loaded when crossing module boundaries, but addresses become longer which may also add to the run-time of a program. Either method could have been implemented on the systems we ported Oberon to, but special hardware support available or compatibility to existing calling conventions had a large impact on our decision about which model to use. Table 5 illustrates this.

Target	Addressing
NS32000	relative
MC68020	absolute
SPARC	absolute
MIPS	absolute
RS/6000	relative

Table 5 Addressing of Global Variables.

Local Variables and Procedure Activation

Local variables and formal parameters are declared inside procedures and their lifetime is equal to the time the enclosing procedure executes. Their allocation is therefore connected to procedure activation which naturally leads to a stack of activation records. An activation record contains the local variables and parameters of an activated procedure, a reference to the previous record, and the address at which execution should continue when the procedure terminates.

These activation records are allocated within a stack delimited by a dedicated register, a stack pointer. Variables can be accessed relative to a second register, the frame pointer. Activation records have a dynamic size if the formal parameter list contains one or more open array parameters passed by value.

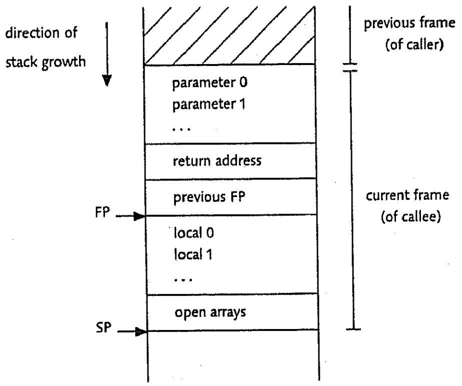


Figure 8 Activation Records.

The stack frames of the various target machines differ in many details. However, their structure is always similar to Figure 8. Actually, the boundary between adjacent frames is not as clear as shown in Figure 8 because the parameter block is accessed from both the caller (for parameter passing) and the callee and therefore constitutes an overlapping area between two adjacent frames.

If registers are used to hold variables or parameters, the register set has to be maintained in a stack-like fashion, too. This means that the register set has to be divided into a number of banks (windows) and that every procedure activation is associated with one of these banks. As procedure calls happen in a last-in/first-out order, the register set acts as a stack of register banks. In order to provide for the passing of parameters in registers, adjacent register banks must overlap to a certain degree. Maintainance of the stack may be either done in software by inter-procedural register allocation or by hardware as implemented on the SPARC. In any case, however, since every register set has a limited size, registers have to be spilled if a stack over- or underflow occurs. On machines with register window hardware, this spilling happens implicitly when switching from one window to another (by a software trap handler on SPARC). On other machines, code has to be generated for it.

A common framework to discuss register usage in different architectures is the subdivision of the register set into caller-saved and callee-saved registers. Callee-saved registers may be used by the caller as if they were local variables, i.e. they are invariant across procedure calls. Caller-saved registers may be changed by the callee, therefore they must be saved (if needed) by the caller. Callee-saved registers are used for allocating local variables. If the variables belong to a procedure that does not call further procedures (a so-called leaf procedure), one can even spare the effort of saving and restoring registers by allocating them in caller-saved registers (see "Target Specific Optimizations" below).

Procedure activation involves both the caller and the callee. The former saves caller-saved registers, assigns the actual parameters to the callee's formal parameters, provides a return address and finally transfers control to the latter. After the callee returns, the caller has to restore caller-saved registers. The callee's actions are described below (see "Procedure Entry and Exit").

Temporary Variables

Evaluating complex expressions requires saving intermediate results in free registers. Such temporary values are placed into caller-saved registers except for the SPARC back-end, where unused callee-saved registers (windowed registers) are preferred.

There are different methods to maintain the set of temporarily used registers. One can use a reference counting scheme to decide when a register becomes free, or one can free registers whenever they are used as source operands in a subsequent operation (unless they are locked explicitly). When a value in a register is required several times (e.g. on method calls, open array index checks), it has to be locked and unlocked explicitly. Counting the number of simultaneous uses of a register is more expensive to implement, but it is worthwhile for architectures with few registers, because registers are freed as soon as possible.

On machines for which it is worthwhile to reschedule instructions in order to increase performance (typically deeply pipelined or superscalar machines, e.g. RS/6000), one should not introduce new dependencies between instructions by reusing the same register unless necessary. This can be achieved by allocating temporary registers in a round-robin fashion which guarantees a maximal distance between two uses of the same register. Note that instruction rescheduling is not performed by any of our code generators, but can be done by a separate peephole-optimizer.

Target	Prefers	Allocation	Deallocation
NS32000	caller saved	stacked	reference counting
MC68020	caller saved	stacked	reference counting
SPARC	callee saved	stacked	after first use or unlocking
MIPS	caller saved	stacked	after first use or unlocking
RS/6000	caller saved	round-robin	after first use or unlocking

Table 6 Strategies for Allocation of Temporaries.

Procedure Entry and Exit

The complexity of the code necessary to maintain the stack of activation records on procedure entry and exit varies strongly, depending on the processor architecture and on the degree of optimization. CISC processors usually provide a special instruction for procedure entry that saves registers, allocates space for local variables, saves the old frame pointer, and sets the new one. Another instruction, symmetric to the first one, is provided for procedure exit. On RISC processors, entry and exit protocols have to be coded with several instructions. An advantage lies in the fact that the code can be optimized and tuned to each procedure. If a procedure does not use callee-saved registers, for example, then the code for saving them need not be present, or if a procedure does not have local variables, then the stack pointer may remain untouched. There is a lot of room here for optimizations that are not possible when using special and complex instructions that perform often unnecessary operations.

However, since the way parameters are passed is part of the machine-dependent calling conventions, there is little freedom left to compiler writers. Indeed, if procedures created by other compilers on the same machine have to be called, then the identical calling conventions must be observed. On CISC machines, all the parameters are usually passed on the stack. The caller pushes each argument separately and copies larger blocks (records and arrays) if necessary.

On RISC machines, the first few parameters are passed in registers. The registers used and their number are defined by the calling conventions. Floating-point and integer values are often treated differently. Blocks are not copied by the caller but by the callee, so that code need not be duplicated. The address of the block to be copied is passed instead of the block itself. The remaining parameters are passed on the stack. Although this method increases performance, it may further complicate the register allocation process. Typically, parameter registers are allocated in the caller-saved area. This reduces the number of available registers for the evaluation of the actual parameter expressions. Architectures with register-windows allow to overlap parts of two subsequent register windows, making this the obvious area to pass register parameters. For more details on SPARC register windows and code generation see [20].

Decrementing the stack pointer between each parameter push would be a waste of code and time. One should decrement it only once for all parameters at each procedure call point, but a better solution is to do it once on entry of the caller procedure for all parameters of all its callees. The activation record of the caller procedure is thus extended by a call area where the arguments of callee procedures will be stored. The size of this area is the maximum needed over all callees. Hence, the size of the frame is not known before all calls inside the procedure have been compiled. The entry code can be patched after the compilation of the entire procedure. Using this technique, only one decrement of the stack pointer suffices for allocating local variables, reserving space for saved registers, and reserving a call area. But in one case, the frame size cannot be predicted at compile-time, i.e. when an open array is passed as a value parameter. One has to create a frame excluding the size of the open array first, and to later expand the frame when the array is copied.

Another possible optimization is to eliminate the frame pointer. When the activation frame is allocated at once and its size is statically known at compile-time, the offset between the frame pointer and the stack pointer is constant and all variables can therefore be addressed relative to the stack pointer. However, this optimization is impossible when open arrays are passed as value parameters to the procedure, which is very rare in average programs.

PROC P()	NS32000	MC680200
register params	none	none
frame alloc	stepwise	stepwise
fp elimination	no	no
entry patch	no	no
min. frame size	8	8
entry	ENTER [ ], vsize	LINK.W FP, #vsize
exit	EXIT [ ] RET psize	UNLK FP RTD.W #psize

PROC P()	SPARC	MIPS	RS/6000
register params	6 int, 0 float	4 int, 2 float	8 int, 13 float
frame alloc	once	once	once
fp elimination	no	yes	no
entry patch	yes	yes	yes
min. frame size	96	0	64
entry	save sp, -size, sp	[ADDIU sp, sp, -size] {SW rx, offsetx(SP)}	stu SP, SP, -size stm Rx, offsetx(SP) cal FP, offsetf(SP)
exit	jmpl ra + 8, g0 restore g0, g0, o0	{LW rx, offsetx(SP)} [NOP] JR ra [ADDIU sp, sp, size]	lm Rx, offsetx(SP) l SP, 0(SP) bcr

Table 7 Procedure entry and exit.

Expression Evaluation

The general structure of expression evaluation is the same for all back-ends. The differences lie in the mapping of the syntax tree nodes to code patterns of the target machine. Some architectures use two-address operations, i.e. an operation takes two source arguments, one of which is implicitly taken as the destination, while others use three-address operations, in which an operation takes two source and one explicit destination arguments. Besides the number of arguments, there is an even more important difference in the addressing modes allowed for the arguments. Some machines restrict the operands to be registers or immediate values (so-called load-/store-architectures) while others allow arbitrary addressing modes to be used for each operand. As a rule of thumb, modern RISC architectures use a load-/store-approach and three operand instructions, whereas the common CISC machines use arbitrary addressing modes and two operand instructions.

It turned out that the RISC approach simplifies code generation in general, as it is more regular and there are less addressing modes to observe. It is also easier to generate the binary instruction patterns for these machines.

As explained in "Code Generation Overview" above, the various back-ends differ in the treatment of the destination argument as shown in Table 8. The SPARC-Oberon approach was to pass a third item to the OPC procedures that defines the destination argument. In case of a register item this register is taken as the destination, otherwise a new register is assigned. Therefore, the third item is interpreted merely as a hint for the code generation procedures. The other variant taken for MIPS and RS/6000 is to pass a register number as a hint and to return the result in one of the other items. The three item approach actually passes more information and allows to optimize some (rare) patterns, whereas the latter approach is somewhat more efficient. The example (Add) contained in Table 8 refers to an OPC procedure that emits the code patterns for addition of two operands.

Target	Architecture	operands	Destination	Example	Explanation
NS32000	CISC	2	implicit	Add(x, y)	result into x
MC68020	CISC	2	implicit	Add(x, y)	result into x
SPARC	RISC	3	item	Add(x, y, z)	result into z, hint z
MIPS	RISC	3	register nr.	Add(x, y, n)	result into x, hint n
RS/6000	RISC	3	register nr.	Add(x, y, n)	result into x, hint n

Table 8 Treatment of destination.

When looking at the code generated for the Oberon statement *a := b + c*, we notice that if all operands are in registers, for three operand RISC machines only one instruction is generated, which typically executes within one machine cycle. For two operand CISC machines three instructions are generated executing in several cycles. This compensates for the additional load and store instructions sometimes necessary in a load/store architecture and leads to reasonable code density. In Table 9, operands are accessed frame-pointer-relatively for NS32000 and MC68020 and inside registers on the RISC machines. Note that even if operands are registers and optimizations are performed, two-address machines need at least two instructions in this particular example.

Target	Bytes	Instructions
NS32000	9	MOVD b(FP), R7 ADDD c(FP), R7 MOVD R7, a(FP)
MC68020	12	MOVE.L b(FP), D7 ADD.L c(FP), D7 MOVE.L D7, a(FP)
SPARC	4	add b, c, a
MIPS	4	ADD a, b, c
RS/6000	4	a a, b, c

Table 9 Code generated for "a := b + c".

Interlocks

All three RISCs considered here have an instruction pipeline and a Load/Store architecture. An operand loaded from memory is not immediately available in a subsequent instruction (load delay). Pipeline interlocking is detected by the SPARC and RS/6000 processors, but not by the MIPS. This introduces additional complexity in the MIPS code generator, which has to detect data dependencies and to insert NOPs when necessary. Note that this is not an optional optimization, but a requirement to generate correct code.

The SPARC and MIPS have delayed branches, i.e. the branch instruction gets into effect only after the subsequent instruction has been executed. Our back-ends do not fill delay slots with useful code because this introduces dependencies between non-adjacent nodes in the syntax tree. The

MIPS and SPARC back-ends simply insert NOPs, except for pre-coded patterns, e.g. block moves. A separate peephole optimizer can later reorder the instructions and remove NOPs.

Other so-called "reorganization constraints" of instructions referring to the multiply/divide unit introduce some more complexity in the MIPS code generator: these constraints prevent the use of instructions in an order for which the hardware cannot guarantee correct results. Basically, the code generator has to insert enough (sometimes dummy) execution cycles between two critical instructions. Additionally, floating-point compare and subsequent floating-point conditional branch instructions on SPARC and MIPS must be separated by at least one dummy cycle.

Run-time Checks

In order to provide referencial integrity in Oberon, there are some checks necessary at run-time. These include type guards, index checks, and NIL checks. Testing for integer overflow or recognizing divisions by zero are a different class of run-time checks that are either provided by hardware or not, but they have no influence on the referencial integrity of programs. Therefore, the various Oberon implementations don't pay much attention to them. A crucial architectural feature for efficient run-time checks are conditional instructions which activate an exception handler if a certain condition holds.

The way checks are implemented depends strongly on the features the target architecture provides. Some machines just provide unconditional trap instructions, making it necessary to implement the check by a conditional branch and a trap instruction. Others provide a conditional trap which makes it possible to raise an exception depending on the value of the condition code registers. The most sophisticated support for run-time checks are check instructions that combine a compare and the conditional trap into one instruction, making checks fast without affecting the control-flow of the program.

Moreover, testing for NIL-access can be performed by a memory-management unit, if available. Most machines protect certain memory areas against accesses, which can be exploited for NIL-checks by selecting the address of this area as the value NIL. This is almost always the value 0.

Without going into details of type guards it should be noted that they essentially consist of a comparison of two pointers to type-descriptors (t0, t1, loading not shown). Activating procedure variables closely corresponds to dereferencing pointers and is therefore not shown explicitly.

run-time checks	p(T)	a[i]	p↑.f
NS32000	CMPD t0, t1 BEQ ok BPT guard ok ...	CHECKW R7, bounds, i [FLAG]	–
MC68020	CMP.L t0, t1 TRAPNE.W #guard	CHK.[W/L] #bound, D7	TST.L A4 TRAPEQ.W #niltrap
SPARC	subcc t0, t1, g0 tne guard	subcc i, len, g0 tcc index	–
MIPS	BEQ t0, t1, ok NOP BREAK guard ok ...	SLTIU r1, i, len BNE r1, r0, ok NOP BREAK index	–
RS/6000	t neq, t0, t1, guard	t ugte, i, len, index	–

Table 10 Run-time checks.

Oberon-2 Type-Bound Procedures

Implementing the activation of type bound procedures (method calls) turned out to be only a small extension to the original Oberon compilers. This simply involves an indirection via a method table that defines the procedures bound to the type of the receiving object. The already existing type-descriptors of Oberon objects can be extended to contain the method table. Two load operations are necessary to obtain a method address, followed by a call of a procedure variable. The method table is allocated with negative offsets within the type descriptor. The receiver is passed exactly like any other parameter. There is no distinction between methods and procedures during the compilation of their body. Only the calls are different. Table 11 shows how this is implemented. Calls of overridden methods (super calls) are statically bound and can be treated like ordinary procedure calls. Oberon-2 has not been implemented for the MC68020.

method call	NS32000	SPARC	MIPS	RS/6000
p.Doit	MOVD –4(p), R7 MOVD p, TOS CXPD offset(R7)	add p, 0, o0 ld [o0–4], tag ld [tag+offset], m jmpl m, o7 nop	LW tag, –4(p) OR r4, r0, p LW m, offset(tag) NOP JALR ra, m NOP	oril R3, p, 0 l tag, –4(R3) l R0, offset(tag) mtspr CTR, R0 l TOC, offset+4(tag) bccl l TOC, 20(SP)

Table 11 Implementation of a method call.

Back-End Particularities and Target Specific Optimizations

The general strategy throughout all back-ends was to use context-free code generation. However, some target-specific optimizations are possible with little additional effort. Note that generating reasonable code such as using arithmetic shifts for multiplication and division by powers of 2 is not considered an optimization, but standard practice.

The only optimization performed by the SPARC back-end is a cache for global variable access. Whenever the first global data address (static base) of a module is loaded into a callee-saved register within the first basic block of a procedure, this register is used as static base until the end of the procedure. It turned out, that this optimization actually reduced the code size of the compiler itself, in spite of a longer source program. A simple peephole optimizer running as a separate tool has been implemented to utilize delay slots, eliminate load interlocks, and eliminate loads immediately following stores at the same address. Although nearly all delay slots can be filled with useful instructions, the average code improvement is below 5%. It is therefore hardly justified to complicate the compiler by these optimizations.

If the number of callee-saved registers used in a procedure cannot be predicted at the time the procedure prologue is compiled, one has to fix up the entry code at the time it is known. This can happen when callee-saved registers are also used for temporary values after the allocator ran out of caller-saved registers, or when inter-procedural register allocation is performed, as implemented in the MIPS back-end. If there is an instruction that saves a set of registers at once, the fixup is simple, but if individual store operations have to be emitted for each register, also the size of the entry code changes, making it necessary to move code at fixup time.

Some optimizations performed by the same back-end, i.e. register allocation in caller-saved registers, entry and exit code optimization, frame pointer elimination, have been briefly described above. Figure 9 shows two procedure activation frames on the MIPS in more detail.

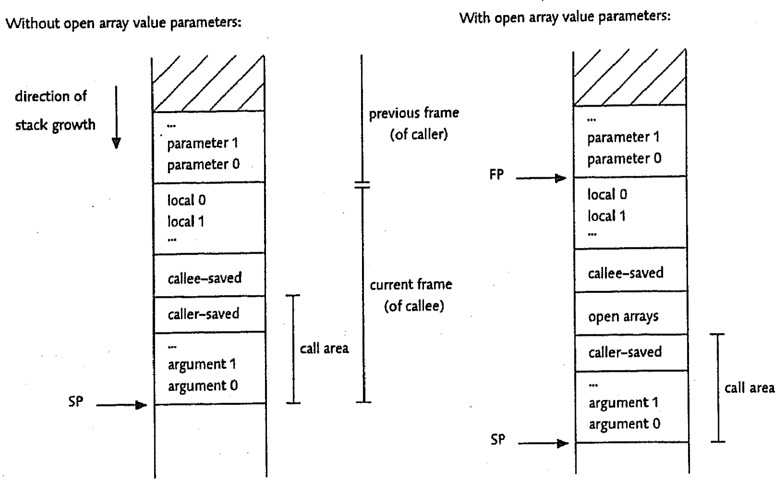

Without open array value parameters:
With open array value parameters:
direction of stack growth
...
parameter 1
parameter 0
local 0
local 1
...
callee-saved
caller-saved
...
argument 1
argument 0
SP
call area
previous frame (of caller)
current frame (of callee)
FP
open arrays

Figure 9 Activation Records on the MIPS.

The size of the left frame is known at compile-time, allowing all local variables to be addressed relative to the stack pointer. By this, the cost of setting up a new frame pointer and saving and restoring the previous one can be spared. The stack pointer is simply decremented by the frame size in the prologue and incremented by the same amout in the epilogue. On the right, a callee-saved register is allocated for the frame pointer, the previous one being saved and restored like other callee-saved registers.

The size of each area of the activation frame is minimized. In the extreme case of a leaf procedure, without parameters and without local variables, the frame size is zero, there is no entry code and only one instruction (JR ra) constitutes the exit code.

The *caller-saved* area is the location where temporary registers that live across procedure calls must be saved. This occurs when function calls are part of expressions or when actual parameters are themselves results of function calls. The MIPS back-end reduces the number of such save/restore operations by modifying the evaluation order of actual parameters.

A sophisticated register allocation strategy, as described in [21], has been implemented in the MIPS back-end. The idea is to use information about inter-procedural register usage in the register allocator, making it possible to neglect the register saving conventions and to make all registers work in the caller-saved mode. The allocator tries to avoid re-use of registers already used in called procedures, hence sparing the cost of save/restore at the call point.

Let's take an example: Procedure P0 calls procedures P1 and P2. All three procedures are not exported. P1 and P2 have been compiled already, hence, the sets of registers R1 and R2 modified by P1 and P2, respectively, are known. P1 and P2 do not save these registers on entry, since all registers work in the caller-saved mode. When P0 is to be compiled, the allocator will not use registers of R1 or R2 for variables of P0. In this example, no register saving is necessary.

Procedures have to be processed in depth-first traversal of the call graph. Before compiling each procedure, the abstract tree of its body is traversed to collect all the calls and corresponding register usage information. In case of recursion or external calls, the information is not present and the default register saving conventions are applied. This inter-procedural register allocation is not always performed, but controlled by a compiler-option. The effect of this optimization is to move the register saves and restores upwards in the call graph. The average number of callee-saved registers to be saved on entry of a procedure (return address, stack pointer and frame pointer included) is reduced from 3.63 to 3.11, but the average number of saves at call points is increased from 0.06 to 0.5. The resulting speedup is strongly dependent on the call graph of the module. It can range from -2% (slower!) to +15%. The results could be improved if the register allocator had more information about the use of variables (e.g. live range of variables), allowing the use of the same register for different variables (e.g. with graph coloring techniques) [22].

Intermediate-level variables are not accessed via a conventional static link or a display, but a different scheme using a global display as described in [23] has been implemented. Like for the SPARC, a separate peephole optimizer has been made available.

The RS/6000 back-end reduces the amount of state saved in procedure prologues, particularly the link register and the global data pointer in leaf procedures. It also reorders the sequence of operations in the prologue and the epilogue in order to reduce pipeline interlocks.

Since the RS/6000 does not provide instructions to load bytes with sign-extension, a sequence of an ordinary byte load followed by two shifts has to be generated when a SHORTINT is read from memory. The back-end delays the generation of the shifts until the sign-extended value is required, avoiding the shifts altogether in many common cases like assignments of SHORTINT variables.

The NS32000 and MC68020 back-ends can also allocate variables in registers. Since there are no callee-saved registers but only 8 caller-saved registers, the programmer defines himself (with a compiler-option) the maximal number of registers to be used for local variables. Some benchmarks have shown that the benefit of allocating local variables of non-leaf procedures in registers is lost when these registers have to be saved across procedure calls. This might be improved by separating the 8 registers in two non-empty sets of caller and callee-saved registers, but the code generator would probably run out of registers. Consequently, the optimization is performed for leaf procedures only.

We do not optimize the code for the NS32000 and the MC68020 with the exception of register allocation for variables as discussed above. This does not mean that the code is bad. Special attention has been taken to use the many addressing modes of the processor as well as possible. This implies delayed code emission and, hence, complex "Items" truly reflecting the processor architecture.

The MIPS, RS/6000 and NS32000 back-ends also recognize and optimize patterns of the form "designator := designator op operand", in which the two designators are the same. This is actually a target independent optimization. Note that a target independent optimization phase may be applied to the tree directly. This phase improves the generated code for all target machines without any change in any back-end.

Target	gen. purpose	data	address	float. point	others
NS32000	8 (8/0)	–	–	8 (8/0)	FP, SP, SB, MOD, CC
MC68020	–	8 (8/0)	8 (8/0)	8 (8/0)	CC, FPCC
SPARC	32 (16/16)	–	–	32 (32 / 0)	Y, CC, FPCC
MIPS	32 (17/11)	–	–	32 (20/12)	LO, HI
RS/6000	32 (10/19)	–	–	32 (13/19)	8 CC, MQ, CTR

Table 12 Number of register in each class (caller-saved / callee-saved).

Unfortunately, there are different classes of registers on our machines, which complicates the task of writing a compiler. Table 12 shows the number of registers in each class (general purpose, data, address, floating-point, special purpose) and the subdivision into caller-saved and callee-saved sets which is a software convention (except in the presence of register windows like on the SPARC).

The MC68020 differentiates between data (arithmetic) registers and address registers. Most operations and addressing modes are applicable to only one of these register types, which complicates the code generator considerably. Except for the NS32000, the other processors use general purpose registers as frame pointer and stack pointer, instead of dedicated ones. The RISCs have special registers for multiplication and division (Y / LO, HI / MQ). The MIPS is the only one using general purpose registers as condition code registers.

Measurements

Table 13 presents some measurements to allow a quantitative comparison of different architectures and different programming systems. It shows the compiler sizes of the various Oberon implementations to give some feeling of the relative back-end complexities, code density, and overall system performance. Source code size is measured in number of statements rather than in number of lines, because this reflects the complexity more accurately. The statement count for the front-end differs because module OPM defining the target machine characteristics has been included, and because on the MC68020 the Oberon-2 extensions have not been implemented. Code size is given in bytes and self-compilation time is measured in the effective number of seconds a user has to wait for the task to complete.

The following machines have been used for timings:

NS32000:	25 MHz 32532 based Ceres-2 computer.
MC68020:	40 MHz 68030 based Macintosh IIfx.
SPARC:	20 Mhz SPARCstation 1.
MIPS:	25 MHz R3000 based DECstation 5000.
RS/6000:	25 MHz RIOS based Model 530.

	NS32000	MC68020	SPARC	MIPS	RS/6000
front end source (statements)	3670	3287	3647	3689	3705
front end code (bytes)	38108	49984	70820	93912	74440
back end source (statements)	2337	2198	2339	3066	4234
back end code (bytes)	26824	40364	44408	79856	69820
self-compilation time (seconds)	33.0	8.9	9.0	7.0	7.9

Table 13 Oberon Compiler Details.

Table 14 shows a comparison of Oberon compilation and execution times with C, based on the Dhrystone benchmark. *cc* means the standard C compiler (without optimizations) available on a machine (MPW C on the MC68020), *cc -O* means the C compiler with maximal optimization level allowing generation of separate object files. All optimizations available in the Oberon compiler have been turned on. Compilation times are given in seconds, and Dhrystones in sec^{-1} (the more the better). C compilation has not been measured for NS32000 because there is no C compiler for the Ceres computer. When comparing compilation times, one should note that for C an additional linking step is required which has not been included here.

	NS32000	MC68020	SPARC	MIPS	RS/6000
Dhrystones Oberon	6677	11338	18300	33829	34768
Dhrystones cc	–	11111	8950	26087	30293
Dhrystones cc –O	–	10526	18500	37037	76306
Compilation time Oberon	1.69	0.33	0.44	0.21	0.23
Compilation time cc	–	4.9	2.2	0.8	1.6
Compilation time cc –O	–	5.1	9.0	1.7	7.7

Table 14 Dhrystone measurements.

THE SYSTEM INTERFACE

In retrospect, it seems logical that the various Oberon systems were implemented on top of existing operating systems on the target machines, with the exception of the original Oberon System on Ceres and on Chameleon, both machines having been developed at ETH. During the early stages of our project, however, it was by no means so obvious that this approach would yield workable systems offering satisfactory performance. The alternative would have been to implement Oberon on the bare hardware of the target machines, based on the Ceres implementation, most of which is written in Oberon and therefore portable.

While raw performance and a convenient user interface seem like the ultimate goal of any operating system implementation effort, most users are in fact willing to sacrifice raw performance in favor of compatibility. Nobody would want to use the Oberon system on their machine if as a tradeoff they had to discard all software already running there. The only solution that pleases end-users is to offer Oberon as a single process running concurrently with other applications under the host operating system, and sharing ressources with these other applications. Consequently, our implementations are all based on the native operating system and display driver of their respective target machine. Table 15 gives an overview.

Target Machine	Processor	Operating System	Display Driver
Ceres	NS32000	Oberon (native)	Assembly language (native)
Macintosh II	MC68020	Macintosh (proprietary)	QuickDraw (proprietary)
SPARCstation	SPARC	UNIX (SunOS)	PixRect (proprietary) or X
DECstation	MIPS R2000	UNIX (Ultrix)	X Window System
RISC System/6000	RIOS	UNIX (AIX)	X Window System
Chameleon	LSI LR33000	Oberon (native)	Oberon (native)

Table 15 Overview of the Target Machines' Processors, Operating Systems and Display Drivers.

The parts of the Oberon system that had to be rewritten for the new target architectures include the module loader, memory management, device drivers, and the file system. These modules constitute the inner core of Oberon. All of the remaining modules can be ported to new systems by recompiling the original Ceres sources for the new target machine. This includes the modules for text management and all user interface modules.

It was easier to port Oberon to Chameleon than to other machines. Since the processor implements the MIPS architecture, it was not necessary to write a new code generator. Hence, the same module loader, memory management and boot-linker as on the DECstation could be used. The file system is the same as on Ceres, only the device drivers had to be rewritten (in Oberon!). A software emulator for floating-point instructions was written in Oberon, since the machine does not implement them in hardware.

Interfacing to Host System Libraries

Our implementations of the Oberon system all offer the same, standardized programming interface to application programs [24]. In addition, they also offer mechanisms for interfacing with the operating system of the host machine.

These mechanisms include access to supervisor calls of the host operating system, and on some machines also access to shared object libraries. Libraries that cannot be loaded dynamically are statically linked to the Oberon Kernel and accessed through a single Oberon Kernel procedure. This facilitates changes in the set of exported library objects by rebuilding the Oberon Kernel without invalidating its interface. None of the Kernel's clients need therefore be recompiled when library functions are added.

Target Machine	Operating System	Supported Supervisor Call Interface	Shared Libraries
Ceres	Oberon (native)	Trap Instruction	n/a
Macintosh II	Macintosh (proprietary)	Arbitrary Instruction Sequence	n/a
SPARCstation	UNIX (SunOS)	Arbitrary Instruction Sequence	yes
DECstation	UNIX (Ultrix)	Arbitrary Instruction Sequence	no
RISC System/6000	UNIX (AIX)	Dynamic Link Library	yes
Chameleon	Oberon (native)	Arbitrary Instruction Sequence	n/a

Table 16 Host System Interface.

The Module Loader

The purpose of the module loader is to load Oberon modules into memory and to resolve references to other modules. In Oberon, modules may be loaded at any time, from which point onwards they may be referenced. It is possible to load modules and activate procedures by their name, without statically linking to them. Since no similar feature was present in any of our target systems, we had to construct our own module loaders and use private object file formats. These differ slightly for the various target machines, but all contain in some form the information listed in Figure 10, which is more than is usually contained in object files on systems that do not support dynamic linking and procedure activation by name.

Header	Module identification and miscellaneous information
Entries	Entry points of exported procedures
Commands	Names and entry points of commands
Pointers	Offsets of global pointers (for garbage collection)
Imports	Names and keys of imported modules
Links	External references that have to be fixed by the loader
Constants	Raw constant data
Code	Raw object code
Types	Data to build type descriptors
References	Reference information (for symbolic debugging)

Figure 10 Components of an Oberon object file.

Loading a module is a recursive process: After reading the object file into memory, all imported modules are looked up in the global list of loaded modules and if they are not found, they are loaded into the system as well. At this stage, key checks are performed that allow the detection of modules compiled against older versions of the modules they import. The key of a module serves as a version stamp of the interface the module provides. At compilation time, a new key is selected for the compiled module whenever the interface has been changed. Moreover, the key of the module as well as the keys of all imported modules are written to the object file. If the interface of an imported module changes later, the loader will be able to detect this by comparing the keys.

If a module cannot be loaded or if one of the key checks between modules fails, the loader returns and all partly loaded modules are discarded. If all imported modules are loaded, open references to them are resolved by fixing up the object code in the client module, type-descriptors are allocated and initialized, a module descriptor is inserted into the global module list, and the body of the module is called.

Not only can modules in Oberon be added dynamically at run-time, they can also be removed from the system which is necessary whenever the compiler has generated a new version of a module that is already loaded and the user would like to utilize the new version. Modules may only be unloaded when they are not imported by other modules. A reference-counting scheme is used to verify this; cyclic imports are illegal in Oberon.

Obviously, the Oberon loader cannot load itself. Furthermore, the normal loader of the native operating system does not support all of the services required by Oberon. One solution is to compile the loader into the executable format of the target machine. The easiest way of doing this is to write the loader in one of the languages available on the target machine, and the corresponding compiler will do the rest. One can also write a minimal boot-loader that loads a memory image of the Oberon loader, hence avoiding the task of translating the Oberon loader into another language. Such a memory image file is produced by an Oberon tool, the boot-linker, also written in Oberon. On Ceres and on Chameleon, the boot-loader is burned into a ROM and activated after a cold start. On Chameleon, even the boot-loader is written in Oberon, thanks to a tool linking Oberon object files for MIPS into a binary image to be burned into a ROM.

The loader (or the boot-loader for DECstation) constitutes an executable program that is called from the native operating system when starting Oberon. Table 17 lists the implementation language for the various Oberon loaders.

Target Machine	Boot-Loader Implementation	Loader Implementation
Ceres	NS32000 Assembly Language	Oberon
Macintosh II	–	MC68020 Assembly Language
SPARCstation	–	Modula-2
DECstation	C	Oberon
RISC System/6000	–	C
Chameleon	Oberon	Oberon

Table 17 Implementation Language of the Loader.

Memory Management

The main issue in memory management for Oberon is the automatic garbage collector [25]. The method used in all implementations is mark-and-sweep collection. For performance reasons, the garbage collector and the memory allocator have been written in assembly language for the machines on the low end of the performance spectrum. On the faster machines, this could be done in a high-level language. It is worth noting that it was possible to write this service which is at the very bottom of the Oberon system in Oberon itself and, furthermore, in a portable way. Indeed, the same garbage collector is running on the DECstation, on the IBM RISC System/6000 and on Chameleon.

Some enhancements to the Ceres implementation were made. Some of our implementations support running the garbage collector at any time, not just between the execution of commands (see Table 18). The compiler generates a list of global root pointers but no information about local pointers. The stack is examined to find potential pointers, which are then verified by examining the heap.

The original garbage collector employs type-descriptors containing a list of all pointer offsets within an object. This is not sufficient for a complete implementation of Oberon-2, which allows the declaration of arrays with a variable number of elements. One either has to restrict the base types of these arrays to non-pointer types or support the traversal of blocks containing a variable number of objects of some basetype in the garbage collector. The garbage collector then iterates across all elements in the array.

Target Machine	Implementation of Garbage Collector	Stack Traversal	Array Iteration
Ceres	NS32000 Assembly Language	No	No
Macintosh II	MC68020 Assembly Language	No	No
SPARCstation	Modula-2	Yes	No
DECstation	Oberon	Yes	Yes
RISC System/6000	Oberon	Yes	Yes
Chameleon	Oberon	Yes	Yes

Table 18 Implementation Language of the Garbage Collector and Collection Strategies.

Device Drivers

The devices supported by the standard Oberon system on Ceres are a bitmap display, a three-button mouse, a keyboard, an RS-232 serial line, and a network interface. The latter is used for accessing file, mail, and print services. In our implementations running under different host operating systems, these device drivers had to be implemented so that other processes running concurrently on those systems were not disturbed by the emulated Oberon system.

On some of the target systems, mouse and keyboard input are received as events and deposited into so-called event queues, while on others direct polling of the hardware is possible. On the Macintosh the mouse must be polled even though the Macintosh Operating System supports mouse events in principle, because two Oberon mouse buttons need to be simulated by keyboard keys that do not generate events. This is necessary since a standard Macintosh mouse unfortunately has a single button only. The Ceres uses a proprietary network protocol and a proprietary page description language for describing images to be printed at a remote server across the network. These were implemented for the Macintosh which uses the same network hardware as the Ceres. Additionally, PostScript output is offered on almost all implementations.

Target Machine	Mouse Polling	Keyboard	Network	Printer
Ceres	direct	Interrupt to Buffer	CeresNet	CeresPrint
Macintosh II	direct	Event Queue	CeresNet	CeresPrint or PostScript
SPARCstation	direct	Device Buffer	n/a	PostScript
DECstation	Event Queue	Event Queue	n/a	PostScript
RISC System/6000	Event Queue	Event Queue	n/a	PostScript
Chameleon	direct	Interrupt to Buffer	CeresNet	CeresPrint

Table 19 Device Driver Implementation Strategies.

The display interface could be mapped in a straightforward way onto calls to the underlying display systems (see Table 20). When Oberon is started, a window is opened that corresponds to the Oberon screen. Oberon sees this window as its display and does not open any other window nor does it use any of the windowing services of the underlying operating system. Calls to services that set dots, fill areas or copy patterns into the Oberon window are used to simulate the bitmap display present on the Ceres workstation.

Oberon uses a single abstract data type "Pattern" to describe raster images. For example, the Caret image is represented in this way. On Ceres, fonts are implemented as a collection of patterns, which are copied to the screen when a character in this font has to be drawn on the screen. In some of our implementations, this method turned out to be too slow for general usage, because the corresponding display drivers can draw characters much faster when string drawing operations are used. For this reason, we differentiate between Character Patterns and Non-Character Patterns. The Oberon fonts have been translated to the format native to the target operating systems and caching mechanisms allow us to take advantage of the string drawing routines. This optimization has increased the speed of screen updates by at least a decimal order of magnitude.

Target Machine	Fonts	Non-Character Images	String Caching
Ceres	Collection of Patterns	Patterns	no
Macintosh II	Macintosh Font	BitMaps	yes
SPARCstation	Collection of PixRect	PixRects	no
DECstation	Single Pixmap + X Font	Pixmaps	yes
RISC System/6000	Single Pixmap + X Font	Pixmaps	yes
Chameleon	Collection of Patterns	Patterns	no

Table 20 Representation of Oberon Patterns.

File System

Surprisingly, it turned out that the simple Oberon file system could not be mapped easily to the seemingly more powerful file systems of the target machines (see Table 21). In Oberon, there is a clear distinction between file services and directory services, whereas on all of our target systems these two very different concepts are not well-separated on the level of system calls. When a new file is created in Oberon, no entry is made into the directory, making the file anonymous. A name can be entered into the directory at any time, which makes the file permanent on the disk and possibly shadows an existing file with the same name. This behaviour is not directly supported by any of the target systems and had to be implemented by using temporary names for anonymous files, which are renamed when a name is registered in the directory.

Target Machine	File System	Number of Open Files
Ceres	Oberon (native)	unlimited
Macintosh II	Macintosh (proprietary)	small
SPARCstation	UNIX (SunOS)	configurable
DECstation	UNIX (Ultrix)	configurable at Kernel generation
RISC System/6000	UNIX (AIX)	configurable
Chameleon	Oberon (native)	unlimited

Table 21 File System.

Moreover, Oberon distinguishes between the abstract data type "file" and the access mechanism that is used to read or modify it, which maintains a file position. This facilitates reading and writing of the same file at several different positions with full synchronisation. All of our target systems, on the other hand, associate a position with each open file and do not synchronize accesses to multiply opened files. This problem could be circumvented by not opening a file more than once, by maintaining a table of open files and by implementing the buffering and synchronization mechanism in Oberon. In order to avoid unnecessary system calls, newly created files are opened lazyly, i.e. they exist in main memory until they reach either a certain size limit or are explicitly registered. As the Oberon system has no concept of physically closing files, the garbage collector had to be extended to return file descriptors of unused files to the host file system, due to stringent restrictions on the total number of open files. All these problems significantly contributed to the complexity of the interface module.

SUMMARY AND CONCLUSION

In spite of being conceived as the native operating system for one specific workstation, Oberon has proven itself to be truly portable. Due to its simplicity and orthogonality, each of the implementations described here could be completed in a single man-year or less. Oberon has also demonstrated that it is an ideal environment for software creation, as all implementations were cross-developed under an existing implementation of Oberon. We also achieved the goal of genuine portability of programs; Oberon programs developed on one system can be compiled and executed on any of the others without modification.

Looking back, we are pleased with the outcome of our project, and very grateful that we were given the opportunity to participate in it. Few engineers ever get a chance in their lives to delve into the deeper mysteries of an operating system, as we have in the course of our enterprise. Surprisingly, however, there were no real mysteries to be found in Oberon, but a lot of good and solid engineering.

ACKNOWLEDGEMENTS

We are indebted to N. Wirth for initiating and guiding our project and for his numerous suggestions and comments.

We are grateful for the support of the Swiss Federal Institute of Technology in Zurich, and its Institute for Computer Systems in particular, which employed us as research assistants and provided the infrastructure under which these implementations were developed.

The implementations for Digital Equipment's DECstation and IBM's RISC System/6000 have been supported financially by Digital Equipment's Systems Research Center in Palo Alto, California, USA, and by the IBM Research Laboratory in Rüschlikon, Switzerland, respectively. We gratefully acknowledge these contributions.

REFERENCES

1. H. Eberle. *Development and Analysis of a Workstation Computer.* ETH Zurich, Dissertation No. 8431 (1987).
2. N. Wirth. *The Programming Language Oberon.* Software-Practice and Experience, **18**, 671–690 (1988).
3. N. Wirth and J. Gutknecht. *The Oberon System*. Software-Practice and Experience, **19**, 857–893 (1989).
4. R. Crelier. *OP2 - A Portable Oberon-2 Compiler.* Proc. Second International Modula-2 Conference, 58–67 (1991).
5. National Semiconductor Corporation. *Series 32000 Instruction Set Reference Manual.* 1984.
6. Motorola Inc. *M68000 16/32-Bit Microprocessor Programmer's Reference Manual.* Prentice Hall, Englewood Cliffs, New Jersey, 1979.
7. Apple Computer, Inc. *Inside Macintosh.* Addison-Wesley, Reading, Massachusetts (1985ff).
8. SUN Microsystems. *The SPARC Architecture Manual.* Revision 50, August 1987.
9. SUN Microsystems. *The SPARC Papers.* SunTechnology, Summer 1988.
10. G. Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, 1987.
11. International Business Machines Corporation. *IBM RISC System/6000 Technology.* Order Number SA23–2619. 1990.
12. International Business Machines Corporation. *RISC System/6000*. IBM Journal of Research and Development, **34** (1), January 1990.
13. Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.
14. Oliver Jones. *Introduction to the X Window System*. Prentice Hall, 1989.
15. H. Mössenböck and N. Wirth. *The Programming Language Oberon-2*. Structured Programming, **12**, 179–195 (1991).
16. B. Heeb and C. Pfister. *Chameleon: A Workstation of a Different Colour.* submitted for publication.
17. LSI Logic Corporation. *LR33000 MIPS Embedded Processor User's Manual.* 1990.
18. J. Gutknecht. *Compilation of Data Structures: A New Approach to Efficient Modula-2 Symbol Files.* Departement Informatik, ETH Zürich, Report No. 64 (1985).
19. M. Franz. *The Rewards of Generating True 32-bit Code.* Sigplan Notices, **26** (1), 121–123 (1991).
20. J. Templ. *Design and Implementation of SPARC-Oberon*. Structured Programming, **12**: 197–205 (1991).
21. F. C. Chow. *Minimizing Register Usage Penalty at Procedure Calls.* Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, 85-94 (1988).
22. F. C. Chow and J. L. Hennessy. *The Priority-Based Coloring Approach to Register Allocation*. ACM Transactions on Programming Languages and Systems, **12** (4), 501–536 (1990).
23. B. Heeb and C. Pfister. *On Intermediate Variables and Local Procedures as Parameters.* Structured Programming, **12**, 39–42 (1991).
24. M. Reiser. *The Oberon System, User Guide and Programmer's Manual.* Addison-Wesley, 1991.
25. C. Pfister (ed.). *Oberon Technical Notes.* Departement Informatik, ETH Zürich, Report No. 156 (1991).

NOTICE

The implementations of the Oberon System described in this paper are available without fee from ETH. They can be retrieved via anonymous FTP from the host *"neptune.inf.ethz.ch"* (Internet Address 129.132.101.33). Full documentation is included in machine-readable form.