Software Portability by Virtual Machine Emulation

by

Stefan Martin Vorkoetter

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1989

Abstract

The proliferation of diverse computer architectures has resulted in an increased need for portable software, but many portability techniques incur a performance penalty. This penalty can be reduced by restricting the range of architectures to which a program must be portable, at the risk of limiting portability to new hardware as it becomes available.

In this thesis we examine a technique for making a program portable to architectures for which it was not intended, without affecting its performance on other architectures. The technique used is virtual machine emulation, whereby an ideal (for the program) architecture is emulated by another program on an incompatible system. By designing the virtual machine carefully, performance loss due to emulation can be minimized, and the program to be ported can be recompiled for the virtual machine without modification.

Acknowledgements

Although I am the sole author of this thesis, it would not have been possible for me to write it without the help and support of a great many people.

First, I would like to thank Dr. Gaston H. Gonnet, for hiring me as a programmer for the Symbolic Computation Group during the summer of 1986, for assigning to me the project of porting Maple to the IBM Personal Computer, for supporting and encouraging the unorthodox approach that I took, and for being my supervisor during my graduate studies.

I would also like to thank Dr. Keith O. Geddes, Mr. Mike B. Monagan, Mr. Benton L. Leong, Mr. Greg J. Fee, and the many other members of the Symbolic Computation Group at the University of Waterloo for providing input into this project, and answering my many questions about the Maple system.

One of the greatest encouragements in the course of this project has been its successful application to the wonderful Ceres workstation. This would not have been possible without the support of Professor Jürg Gutknecht and Professor Niklaus Wirth. In this vein, I would also like to thank the graduate students at the Fachgruppe Computer-Systeme of the Institut für Informatik at ETH Zürich for their advice and assistance. I would especially like to thank Mr. Clemens Szyperski for bailing me out when I accidentally destroyed the file system on my workstation, and Mr. Cyliax, Mr. Noack, and Mr. Wakefield for providing excellent technical support.

Finally, I would like to thank my wife, Lori Albrough, without whose unfaltering support and encouragement I would never have completed this thesis, my cats, Hannah and Merlin, who patiently sat on my lap as I typed, and my parents, who kept asking me when it would be done, so that one day I could say, "It is done."

To the Empress of the Universe

Table of Contents

Introduction 1

1.1. History 1

1.2. The Maple Machine 3

1.3. Organization of the Thesis 3

Methods of Software Portability 5

2.1. What is Software Portability? 5

2.2. Factors Affecting Portability 5

2.2.1. Programming Languages 5

2.2.2. Operating Systems 9

2.2.3. Machine Architecture 10

2.3. A Hybrid Approach 13

The Portability of Maple 15

3.1. Portability to 16 Bit Machines 16

3.2. Portability to the IBM Personal Computer 17

The Maple Machine 22

4.1. History 22

4.2. Goals of the Maple Machine 23

4.3. Design 24

4.3.1. The Programmer's Model 25

4.3.2. The Evaluation Stack 29

4.3.3. The Return Stack and Procedure Calls 30

4.3.4. Instruction Set 32

Constants 33

Arithmetic 33

Bitwise 34

Boolean 34

Comparison 35

Stack Operations 36

Memory Access 37

Control Flow 38

Hybrids 44

4.3.5. Intrinsic Functions 45

Development Tools 54

5.1. The Compiler 54

5.1.1. The Small-C Compiler 54

5.1.2. Shortcomings of the Compiler 54

5.1.3. Changes to the Compiler 55

5.1.4. Compiler Bugs 56

5.1.5. Preliminary Testing 56

5.2. The Assembler/Linker 57

5.2.1. Overview 57

5.2.2. Why Link at the Assembly Level? 58

5.2.3. Instruction Format 59

5.2.4. Assembler Directives 60

5.2.5. Files Produced 68

The First Implementation 70

6.1. Overview of the Interpreter 70

6.2. Implementation Strategy 70

6.3. Initial Debugging 72

6.3.1. Some Small Test Programs and Results 72

6.4. Getting Maple Running 74

6.4.1. Debugging Facilities 75

6.4.2. Using the Assembler's Map and Correlation Files 78

System Optimization 80

7.1. Improving the Compiler 81

7.1.1. Maple Specific Functions 83

7.1.2. Improved Function Calling 83

7.1.3. Other Optimizations 85

7.2. An Assembler Peephole Optimizer 86

7.2.1. Why Optimize in the Assembler? 86

7.2.2. The Code Burst Table 87

7.2.3. The Fast SWITCH Construct 88

7.3. Improving the Interpreter 89

7.3.1. Representation of Memory 91

7.3.2. Reducing Overhead 91

7.4. Adding New Instructions 93

7.4.1. Generating Instruction Profiles 94

7.4.2. Analysis 97

7.4.3. Modifying the System 99

7.4.4. Iterative Improvement 100

7.4.5. New Instructions 102

7.4.6. Other Optimizations 104

7.5. Some Performance Figures 106

A Real World Example: The Oberon System 107

8.1. A Brief Introduction to Ceres and Oberon 107

8.2. Obstacles for the Oberon Port 109

8.2.1. The User Interface 109

8.2.2. The File System 110

8.2.3. The Oberon Programming Language 111

8.3. Porting the Maple Machine 111

8.3.1. The CFileIO Module 111

8.3.2. Writing the Interpreter 112

8.3.3. Intrinsic Functions 113

8.3.4. Hiding the User Interface 115

8.3.5. Running Maple Under Oberon 119

8.3.6. Porting the Library 120

8.4. Performance 122

Future Directions 124

9.1. The IBM Personal Computer Family 124

9.2. New Instructions 125

9.3. A Threaded Interpreter 125

9.4. Global Optimization 126

9.5. Merging the Interpreter and the Maple Kernel 126

9.6. A Hardware Maple Machine? 127

References 128

Related Works 131

List of Figures

Figure 1. The Architecture of the Maple Machine 26

Figure 2. Instruction Execution 29

Figure 3. Procedure Calling Sequence 31

Figure 4. Structure of a SWITCH Statement 43

Figure 5. Structure of a FOR Statement 64

Figure 6. Structure of IF Statements 67

Figure 7. Structure of a FAST-SWITCH Statement 90

Figure 8. Instruction Pair Execution Profile Before Optimization 96

Figure 9. Instruction Pair Execution Profile After Optimization 101

Introduction

The proliferation of diverse computer architectures has resulted in an increased need for portable software. There exist many techniques for the development of software that is portable across architectures, but these may incur a performance penalty. By restricting the range of architectures across which a program should be portable however, this penalty can be minimized. A problem then results when it is desired to port the program to an architecture that is radically different. It is generally not desirable to reduce the performance of the program on the majority of architectures just so that it can be easily ported to a small number of different ones, so an alternate approach is required.

In this thesis, we will examine a software portability technique that is a conglomeration of various earlier techniques. This will allow us to port a large application program to a variety of atypical system architectures. By "system architecture", we refer to both the hardware of the machine, and the structure of the operating system under which the program must run.

1.1. History

In the summer of 1986, a project was started to port the Maple [3] symbolic computation system to the IBM Personal Computer family.

Maple is an interactive program that can solve problems in many areas of mathematics, such as differential and integral calculus, number theory, linear algebra, and group theory. Maple is also a language that allows users to write programs to solve problems that Maple cannot solve directly. The Maple system consists of two parts. The kernel is written in Margay, an extended subset of the C programming language [10], and can perform basic functions such as arbitrary precision arithmetic, differentiation of simple expressions, and interpretation of the Maple programming language. The library is written in the Maple programming language and provides Maple's higher level functionality. Library routines perform such functions as matrix multiplication, evaluation of trigonometric functions, and integration.

Margay is a macro processor, loosely based on the standard C preprocessor but with many extensions. The Maple kernel is written in terms of a very large collection of macros specifying everything from the target machine's word size, to the syntax for declaring initialized variables. By changing the macro definitions, it is possible to have Margay generate code in the C or B [7] (the predecessor to C and successor to BCPL [18]) programming languages, for a variety of machines. When Maple was first implemented, no C compiler was available to its authors. By using Margay, it was possible to compile the early versions of Maple with a B compiler, and then switch to C when it became available.

The standard procedure for porting Maple to a new machine is to modify the macro definitions to reflect the system architecture of the machine, run the Maple source code through Margay, and then compile the B or C language output on the new machine. If the file naming conventions of the new machine's operating system differ from those assumed by Maple, a conversion routine will also have to be written.

This approach to portability had always succeeded in the past, but it did not succeed in the case of the IBM Personal Computer (PC). The PC architecture differs from any architecture to which Maple had been previously ported in several ways. First, all earlier ports of Maple had been to machines with a 32 or 36 bit word size, while the computers in the PC family were all 16 bit machines. The Maple source code makes extensive use of shifting and masking operators to store information in the bits of a word. The other major architectural difference lies in the structure of the address space. In order to be able to address up to a megabyte of memory using only 16 bit registers, the PC's CPU (the Intel 8088) uses a two part address consisting of a segment register and an offset. This conflicts with the assumption of a flat address space that is apparent in Maple's source code. An attempt was made to compile Maple by changing all declarations of integers to long integers (by changing a macro), and coercing all pointer values to a normalized form to mimic a flat address space, but this resulted in a code image three times the size of the code image on a VAX minicomputer. This would not have left enough memory on the PC for Maple to do any useful work.

At this point, the author of this thesis suggested a different approach. Instead of massaging Maple to make it suitable for the IBM PC, perhaps it would be possible to make the PC look more like a large computer.

1.2. The Maple Machine

The Maple Machine is a program that allows for the execution of Maple on any hardware to which the Maple Machine can be ported. It is between one and two orders of magnitude smaller than the Maple kernel, and thus much easier to port.

In this thesis we will discuss the development of the Maple Machine, its portability, and its performance. We will examine other work done in this area and show how the Maple Machine compares in portability and performance.

1.3. Organization of the Thesis

The parts of this thesis are organized such that each concept is explained before it is used. However, it is also meant to be a viable reference for programmers of future Maple Machine implementations, and thus has been broken into sections describing various aspects of the machine.

Chapter 1 is this introduction, giving an overview and history of the Maple Machine project.

Chapter 2 discusses some of the techniques of software portability, and their advantages and disadvantages, while Chapter 3 describes the portability problems with the Maple system.

Chapter 4 describes the details of the Maple Machine architecture, and the design decisions and compromises that were made.

The compiler and other development tools written for the Maple Machine are described in Chapter 5, while Chapter 6 discusses the process of producing the first functioning implementation.

After a working version of the Maple Machine was completed, much work was put into improving performance. The tools and techniques used are described in Chapter 7.

In Chapter 8, we examine the practical problems of porting Maple and the Maple Machine to a new system architecture, the Ceres workstation running the Oberon operating system.

Chapter 9 summarizes the current state of Maple Machine project, and presents some possible ideas for future enhancements.

Methods of Software Portability

In this chapter, we give a basic definition of software portability, and then discuss the factors making it possible. Factors relating specifically to the Maple system are discussed in Chapter 3.

2.1. What Is Software Portability?

Poole and Waite [17] define software portability as "a measure of the ease with which a program can be transferred from one environment to another: if the effort required to move the program is much less than that required to implement it initially, then we say it is highly portable". Almost any program can be transferred to any machine of sufficient size, but if the effort to do so approaches or exceeds the effort required to write the program in the first place, it cannot be considered portable.

2.2. Factors Affecting Portability

There are several factors that influence the portability of a computer program. The choice of programming language is probably one of the most important, followed by the operating system, and the hardware. However, no combination of choices, no matter how well made, can guarantee portability. Some effort will always be required on the part of the implementor to reduce the effort required to port the program to a different system.

2.2.1. Programming Languages

The most important factor influencing portability is probably the choice of implementation language. A program written in assembly language, for example, will only be portable to computers using the same central processing unit (CPU), and probably only under the same operating system. Similarly, a program written in a proprietary high-level language supplied by a specific machine vendor will only be portable to other machines supplied by that vendor. In order to facilitate portability, a language available on all future target machines should be used. This would appear to require some crystal ball gazing, but several languages have survived the rapidly changing world of computer technology and look as if they will continue to do so for quite some time.

After evaluating a language for portability, it is important that the language chosen is suitable for the task at hand. Using an unsuitable language may require the implementor to use many system-dependent "tricks" to attain satisfactory performance, thus compromising portability.

Once a language has been decided upon, one cannot simply write the program using any and all features provided by the language implementation being used for development. A programming language alone will not ensure portability. Many languages have about them a mythical aura of portability, attributed to them over the years by users who keep telling one another that this is so. Lecarme [12] points out that many people state that, "My program is written in FORTRAN, consequently it is portable". The author of this thesis was recently consulted about porting a graphics-intensive educational program from one brand of microcomputer to an incompatible brand and was told that, "It should be completely portable because it is written in C".

There are several reasons why a program written in a high-level language such as FORTRAN or C would not necessarily be completely portable. Many language implementations (of FORTRAN in particular) offer extensions to the standard language. The use of these extensions is a habit that is very easy to slip into, and moving the resulting programs to machines with an implementation of the language that is lacking these extensions becomes difficult. This is compounded by the fact that the extensions that are used the most tend to be the ones that overcome the greatest deficiencies in the standard language.

Programs do not run in isolation; they make use of facilities provided by the hardware of the machine they are running on. These facilities include mass storage devices such as disks or tapes, output devices such as line printers, plotters, or high resolution colour graphic displays, and input devices such as keyboards and mice. A well written library of routines to support these devices can make the programmer's work easier, but unless these routines are available on other systems, they will not contribute to portability. Thus, a truly portable program can make use of only those devices common to all systems.

One further obstacle to portability that should be hidden by programming languages, and is often ignored in introductory programming texts, are the details of the underlying machine's arithmetic. Programming language definitions tend to discuss arithmetic in terms of integers and real numbers, as if the computer actually dealt with these quantities. In actual fact, most computers deal with integers modulo N, where N is usually a power of two. Real numbers are represented by floating point approximations, which are adequate for most purposes, but can introduce severe errors in complex calculations. As long as the details of numeric representation are documented, programmers can take them into account. However, programs that consider the details of specific machine representations are inherently non-portable since these details differ from machine to machine. The ideal programming language would of course hide all these details. The Maple language actually does hide these details, but with a performance penalty that makes the language wholly unsuitable for general purpose programming.

Several programming languages are suitable for writing portable programs. The one which is probably considered the most portable by many people is FORTRAN. Why this is so is a mystery to the author, since dialects of FORTRAN are about as numerous as FORTRAN implementations; almost every compiler accepts a different language, even though many claim to follow a standard. In order to write portable programs in FORTRAN, one has to restrict oneself to the subset of the language that is implemented equivalently by all compilers. According to Knuth [11], the CONTINUE statement (effectively a no-op) is the only part of the language that meets this requirement.

There exists a program called PFORT [19], which verifies a FORTRAN program for compatibility with the FORTRAN 66 standard. This program checks for correct usage of constructs such as COMMON (which allow sharing of variables between separately compiled modules), the passing of parameters to subprograms, and accidental use of recursion. Incorrect use of the first two may result in a program that works when compiled with one FORTRAN compiler, but not with another. Recursion is not allowed by FORTRAN at all, but is not detectable by the compiler when two mutually recursive subprograms occur in separate source files. PFORT does not check for portability problems resulting from misuse of arithmetic operators, the selector of a computed GOTO going beyond the number of target line numbers specified, and correspondence of I/O formatting fields (in FORMAT statements) with variables (in READ or WRITE statements). Thus, PFORT can provide some assistance with portability, but can not ensure it.

Pascal is considered by Lecarme and many others to be a very portable language. The author agrees wholeheartedly with this, for several very good reasons. Pascal has a clear and concise defining report, making it straightforward for the implementor of a Pascal compiler to ensure that her implementation agrees with both the letter and the spirit of the language. The few ambiguities in the language [22] definition are mostly of academic interest, and not likely to pose any practical problems. Most importantly, Pascal provides powerful data and control structuring facilities, making it suitable for a much wider range of programming tasks then FORTRAN. Unfortunately, it is lacking in some areas that would make it suitable for large projects. For instance, there is no language defined means of implementing separate compilation.

The C programming language [10] is also considered very portable. Unlike Pascal however, C does not attempt to hide all of the details of the machine from the programmer. It is possible to write extremely portable software in C, but it is also possible to write very machine specific code, such as a driver for a complicated peripheral device. C has all the data structuring and control structuring ability of Pascal, but does not provide the same degree of protection against non-portable use of these facilities. Careful use of the C language and division of programs into machine dependent and independent parts can result in very portable systems level programs. By "systems level", we are referring to facilities such as those typically provided by an operating system, such as file manipulation, I/O control, process scheduling, and memory management, as opposed to applications level programs such as word processors or symbolic computation systems like Maple. However, C's structuring facilities also make it suitable for writing portable applications level programs.

Just as there is a portability verifier for FORTRAN, there is also one for C. The C verifier is called *lint* [21], because it is designed to pick the "fluff" out of C programs. *Lint* serves two main purposes: it detects errors and omissions that the C compiler would pass over, and it attempts to point out portability problems. C compilers have traditionally been written to perform a minimum of error checking, under the assumption that the programmer knows what he is doing, and would run a program like *lint* if he wanted to see pages of warnings. Some of the portability problems that *lint* reports are:

1. Relational comparisons of **char** type variables with zero, since these would be invalid under implementations which treat the type **char** as unsigned.
2. Assignment of **long** type variables to **int** type variables, since these would result in loss of precision on machines where the two are not the same size.
3. Assignment of integer values to pointer variables, since the two are not necessarily compatible.
4. Word alignment of pointer values, since this may be required on some machines.
5. Statements such as "a = - b", since some older compilers will interpret this as "a -= b", while the newer compilers interpret this as "a = -b".

As with PFORT, *lint* does not guarantee portability. It is possible to write a program that passes through *lint* unscathed, but is still not portable, due to implicit assumptions about the number of bits in a word or details of floating point arithmetic.

2.2.2. Operating Systems

The operating system provides the environment under which a program must operate. It provides facilities for creating and using files, reading input from the user, and printing or displaying output. The choice of operating system, and consequently the facilities provided, plays a major role in the portability of a program.

Because an operating system can hide many of the details of a machine's architecture from a program, portability is enhanced. If the program makes use of the hardware only through the facilities provided by the operating system, and these facilities are provided in a device-independent way, then the program can be made portable to any machine that runs that operating system. The portability of the program is then dependent on the widespread availability of the operating system under which it runs.

The operating system supported by the greatest number of architectures is probably UNIX [2,22]. UNIX is a multiuser timesharing operating system originally developed by Ken Thompson and Dennis Ritchie at Bell Laboratories in the 1970s. It is implemented almost entirely in the C programming language, and has been ported to a wide range of machines, ranging from supercomputers such as the Cray 2, to large mainframes such as IBM's 3090, to minicomputers such as the DEC VAX series, down to microcomputers such as the IBM Personal Computer AT. UNIX provides a small number of services to allow programs to make use of files and devices. The interface to most of these services is identical across all machines running UNIX.

By programming in the C language, and using only those operating system services provided by all versions of UNIX, it is possible to write completely portable (among machines running UNIX) application programs. In other words, all that is necessary to move the program to another UNIX system is to transfer the source code and recompile it. By using only the portable UNIX functions, portability is increased to many non-UNIX machines since many of the UNIX functions are simulated by the C libraries on these machines. For example, many programs can be moved to a PC-DOS (the standard operating system of the IBM Personal Computer family) based machine by simply recompiling them with one of the many good PC-DOS C compilers and their UNIX support libraries.

Some attempts have also been made in the development of portable operating systems [4]. The basic premise is to write the operating system in such a way that it is easily ported, and then to write many application programs to run under the operating system. When the application programs are to be ported to a new machine, the operating system will be ported first. This is done by retargeting the compiler of the language in which the operating system is written for the new machine, modifying the machine dependent parts of the operating system appropriately, and then compiling the operating system for the new machine. After the operating system has been ported, the applications can be ported by simply recompiling them with the retargeted compiler. This requires that the application programs deal with the hardware only through the operating system, and do not make use of any machine dependent features.

2.2.3. Machine Architecture

Even programs written in a portable language running under a portable or widely used operating system are not necessarily portable. A program may make implicit or explicit assumptions about the machine it is running on. For example, intermediate results of some computations may fall outside of the range of representable numbers on one machine but not on another. Comparisons of characters for lexicographic ordering may fail due to the use of different character sets on different machines. Thus, portability may still be difficult to attain for some applications.

The ultimate in portability would be achieved by having a portable machine architecture. Ideally, this would mean that all machines have the same CPU architecture, the same peripherals, memory map, and speed. An application designed to run on one machine would run on any machine. This is obviously not feasible, as the many manufacturers would never agree on a suitable architecture. Disallowing variability would also put an end to progress. Some work has been done in this direction however. In the early 1980's, Microsoft developed a standard called MSX. This standard described a microcomputer architecture based on the Z80 CPU chip, the TMS 9918 graphics chip, and a fixed memory map. Programs written for an MSX machine would run on any other manufacturer's MSX machine. A few manufacturers actually built such machines, but they never really caught on.

Rather than actually build all machines with the same architecture, another approach is to simulate such an ideal machine with a program. We will call this simulated machine a *virtual machine*. This is not to be confused with the virtual machine in operating systems such as IBM's VM/CMS. In those types of environment, instructions of the virtual machine are still being executed directly by the real machine; only the peripherals and memory addresses are being simulated. The architecture of the virtual machine closely reflects that of the underlying real machine. Our virtual machine on the other hand is being simulated completely; the underlying real machine is completely hidden by the simulation program. This program could then be run on any real machine, and application programs would then run on this simulated machine. This approach also has some problems.

Since the simulated machine is a program, and this program is to run on any real machine, there is a requirement that it be either extremely portable, or extremely small in size. If the program is non-portable and large, the effort required to move it to a new machine will be large. If the program is portable, the porting effort will be small in relation to the development cost of the applications that are to run on the simulated machine.

If the program is small and simple, both the porting effort and development cost will be small.

Since our simulation program must completely hide the architecture of the physical machine (and its operating system, as we will see later) insofar as this architecture differs from that of our virtual machine, it will necessarily be quite machine dependent. We must therefore make the simulator small, since we cannot make it very portable. Thus, our virtual machine must be very simple and well thought out.

Another problem with simulating an ideal architecture is the performance penalty involved. Instruction sets of most CPU's are implemented in microcode, a low-level hardware oriented sequence of instructions that is interpreted by the hardware inside the CPU. This hardware is very fast, resulting in a high instruction execution rate. In implementing our virtual machine, we are effectively writing microcode at a higher level. Each instruction of our virtual machine is written in terms of several instructions of the physical machine. Therefore, the execution speed of the virtual machine will necessarily be slower than that of a real machine.

Each instruction executed by our virtual machine will have associated with it some minimal overhead in execution time. Even an instruction that does nothing needs to be fetched, and the program counter needs to be incremented. It is very important for performance then that the overhead for instruction fetching and decoding be minimized in our virtual machine implementation. Another way to minimize the percentage of time spent fetching instructions is to make each instruction perform many operations. For example, one could implement an instruction that increments a loop index, checks the termination condition, and performs the appropriate branch. Unfortunately, this conflicts with our earlier requirement that the virtual machine must be simple to implement. Thus, there are compromises to be made in the design of the virtual machine; it must be simple enough to implement on almost any real machine, and it must be fast enough to run programs at a reasonable speed.

An example of the successful application of the portable architecture approach is the Pascal-P system developed at ETH Zürich [15]. Here, a portable Pascal system was desired for student use. The compiler was written in Pascal, and generated code for a hypothetical stack machine. This stack machine is relatively easy to implement, so porting the compiler consists of implementing the stack machine on a new computer and transferring the executable code image of the compiler to the new computer and running it on the stack machine. The compiler can then be used to compile code **on** the new machine. A related project is described, complete with source code for the compiler and the stack machine interpreter, by Niklaus Wirth in "Pascal-S: A Subset and its Implementation" [23]. Performance in either of these examples was not of paramount importance, since the compiler and interpreter would only be used to compile and run small student exercises.

2.3. A Hybrid Approach

As was shown, the choice of programming language, operating system, and machine architecture all contribute to portability. A portable architecture would allow all other portability concerns to be ignored, since the programs would run in the same (in every detail) environment on all machines. Even I/O devices and file system differences could be hidden. Effective use of a widespread language and operating system combination would allow portability to a wide class of machines.

When porting an existing program however, little can be done about the portability of the program itself. The program could be rewritten with portability in mind, but this is only feasible if it is then to be ported to many machines, amortizing the cost of rewriting the program. In some cases however, the program might be almost portable. By this we mean that the original implementation was intended to be portable across a wide class of machines, and we have only to extend that class. The solution we will propose is to emulate those features common to the original class of machines, on the new machines.

To accomplish this, we must identify what it is that makes the program non-portable to the new machines. It is not necessary to hide all differences between the existing and new machines, but only those differences that would affect the operation of the existing program. These might include dependencies on word size, assumptions about addressing, or usage of operating system facilities. We then implement an ideal virtual machine that mimics the operation of the class of machines to which the program is portable, and provides all the operating system facilities which the program uses. This ideal virtual machine is simply another program which is then run on the new machines, under the operating system used by that machine. Our goal is to make it impossible for the program running on the virtual machine to determine that it is not running on a real machine to which it would be portable.

After designing a virtual machine, we must develop all the necessary tools to port programs to this machine. These tools include such utilities as compilers, assemblers, and debugging aids. The development of such tools is a very time consuming endeavour, and is worthwhile only if the application program absolutely must be ported, the application is very large and therefore even more time consuming to rewrite for the new machine, many applications will be ported using these tools, or these tools will allow the application to be ported to many new machines. In other words, this is not a rapid way to port a non-portable program; it is a slow (but still faster than actually porting the program) way to make a large program very portable. It may in some cases actually be the only way a port is possible.

The Portability of Maple

The Maple symbolic computation system was written in Margay. Margay is both a macro preprocessor, and the language accepted by that preprocessor. There is no official definition of Margay; it simply evolved from simpler macro preprocessors, with new features being added by Maple's authors as they were required. The main purpose of Margay is to hide the syntax of declarations used by the underlying language (C or B) from the programmer. Margay also provides a limited form of abstract data typing. For example, many variables in the Maple kernel are of type pointer to pointer to pointer to integer. To declare several such variables, the following declaration is used in C:

```
int ***a, ***b, ***c;
```

Note that the base type, **int** is only specified once, but the levels of indirection must be specified separately for each variable. Margay on the other hand allows the definition of new types (as macros), and then allows declarations of that type to be made. For example, the type ALGEB is used to refer to data structures in the Maple kernel. The definition of ALGEB is pointer to pointer to pointer to integer. The C declaration shown above would then be rewritten thus:

```
LOC( ALGEB ) a, b, c;
```

This can also be accomplished using the **typedef** facility of C, but such a facility does not exist in B and in many early C compilers. By using Margay, only the macro definitions need be changed. All actual declarations can be left unmodified. The kernel of the Maple system, which performs functions such as arbitrary precision arithmetic, simple differentiation, and interpretation of the Maple programming language, is written in terms of these Margay macros and C language control structures.

When Maple was first implemented, no C compiler was available to its authors. Only an implementation of B, the predecessor of C, was available. B has most of the same control structures as C, but lacks the concept of typed data. All variables are considered to be words, and their interpretation is left to the operators that are applied to them. There are no structured types such as the **structs** of the C language (early C compilers also lacked the **struct** facility). Consequently, there are no bit field types either. With only a B compiler at hand, and an intention to use C in the future, Maple's authors wrote the Margay preprocessor and a collection of macros with which to express type declarations and keywords not available in B. Maple was then written in terms of these macros and passed through the preprocessor before compilation by the B compiler. When a C compiler eventually appeared, it was only necessary to change the definitions of the macros so that the preprocessor would transform the Maple source code into C instead of B.

Using these techniques, the authors took great pains to ensure portability to other machines. The initial implementation machine used a 36 bit word size, but the Maple source code only assumed a 32 bit word size since later machines might not have 36 bits per word.

3.1. Portability to 16 Bit Machines

Unfortunately, the portability effort also resulted in the greatest portability problems when it was time to port Maple to a 16 bit machine. Since the Maple source code had to be acceptable to both B and C compilers, only the features of C that were also features of B were used. Macros were used to preprocess simple type declarations (which would be preprocessed out for B), but no attempt was made to make use of C's powerful data structuring facilities. Data structures in Maple consist of a sequence of machine words, the first of which is packed with various information about the structure. In a pure C implementation, this would have been accomplished using the bit field facilities of the language, but this could not be done because B had no such facilities. As a result, the information in the first word of each structure was accessed using the masking (&,|) and shifting (<<,>>) operators along with appropriately defined constants. Thus, the assumption that the word size is at least 32 bits became thoroughly engraved into the code. In order to compile Maple on the IBM Personal Computer, it was necessary to change all declaration of the C type **int** to **long int**. Similarly, all constants used in function calls had to have "L" appended so that they would be passed as long integers (2 words) instead of integers (1 word). Newer compilers conforming to proposed ANSI C standard would alleviate this problem by allowing the use of function prototypes.

3.2. Portability to the IBM Personal Computer

Another assumption made in the Maple source code was also invalidated when attempting to port to the IBM Personal Computer. All machines that Maple had previously been ported to had a single large linear address space. This means that memory addresses begin at 0, and increase by 1 until the last addressable location is reached. As a result, an address is simply an unsigned integer giving the location of a byte or word relative to the lowest such location. The Maple code treats integers and pointer types as completely interchangeable, often assigning two pointer values to two integer variables, and performing a comparison of these two variables at some later point in the code. This is perfectly legal (although somewhat "immoral" in the author's opinion) on machines where the assumption holds.

The CPU of the IBM PC (the Intel 8088 or 80286 [9]) family uses a different addressing scheme. Since all registers are only 16 bits, but the 8088 can address one megabyte of memory (requiring 20 bit addresses), a two part addressing scheme is used. The CPU contains four segment registers, one for code, one for data, one for the stack, and one for the heap. Whenever a byte is addressed, the appropriate segment register is shifted left four bits and added to a 16 bit address offset. The segment register used depends on what is being fetched: code, data, etc. although explicit segment overrides are possible. Addresses are written as two numbers, separated by a colon (:). The first number indicates the segment, while the second indicates the address offset within the segment. For example, the hexadecimal address 0100:0200 represents the physical address 01200. Because of the mechanism used to compute physical addresses, there is more than one logical representation for each address. The addresses 0000:1200, 0080:0A00, and 0120:0000 all represent the same physical address as our first example. For each physical address, there are 4096 distinct representations.

The 80386 CPU, used in the high-end models of the latest generation of IBM personal computers, also has a segmented address space. With this CPU however, segments and offsets are each 32 bits, so that a single segment is 4 gigabytes in size. By restricting oneself to a single segment, one effectively has a single large linear address space. A port of Maple to the 80386 was carried out by simply recompiling Maple on such a machine.

C compilers for the 8088 generate code to normalize addresses before comparing pointer valued objects. A normalized address is one in which the address offset is always between hexadecimal 0000 and 000F. Thus, there is only one normalized representation of each physical address. By treating the two parts of the normalized address as one 32 bit double word, an arithmetic comparison of the two pointer values can be made. However, when a pointer value is assigned to a (long) integer variable, it is not first normalized. If two such values are assigned to two such variables, and the variables are then compared, the comparison can, and probably will, give erroneous results. For example, consider the two physical addresses 01200 and 01201. If the first was represented by 0080:0A00 and the second by 0000:1201, and these representations were treated as long integers and compared, then the first would appear to be larger than the second even though the first address precedes the second. The compiler can only generate code to make correct address comparisons if it knows that addresses are being compared. When comparing two long integers containing addresses, such as in the following C statement,

```
if (i == j) break;
```

the compiler can be coerced into making the correct comparison by using C's type casting facility to cast the integer values into addresses:

```
if ((void *)i == (void *)j) break;
```

This tells the compiler to assume that i and j contain addresses and to perform the appropriate normalizations before making the comparison. The problems with this approach are twofold. First, it would be necessary to examine the entire Maple source code to find all comparisons of integers that contain addresses, and insert the appropriate type casts. Secondly, not all of these comparisons are always comparing addresses; the variables may sometimes contain actual integer values, which normalization would invalidate. This use of variables to hold different types arises from Maple's internal data structures, which are simply declared as an array of pointers. The individual fields are allowed to hold any word sized value however.

The integer size and the representation of addresses are the two major obstacles to generating correct code for Maple on the IBM Personal Computer. An attempt was made to make all the necessary changes to the source code to compile a functioning version of Maple. After about two weeks of effort, Maple was finally compiled, using version 3 of the Lattice C Compiler. The result however was not worth the trouble. The size of the code image was 378 kilobytes, which is nearly three times the size of the 140 kilobyte code image on a VAX 11/780. With a limit of 640 kilobytes of memory available to programs running under PC-DOS, this large code image would not leave sufficient space for Maple to perform any useful work.

The unusually large code image was due to several factors. Performing operations on 32 bit integers on an 8088 is very expensive in terms of time and space. For example, consider the code generated for the statement *i* += *j* where *i* and *j* are declared as local long integers:

```
les ax,[bp-8]     ;load MSW of j into ES and LSW into AX
mov dx,es         ;move MSW of j into DX so DX:AX == j
add [bp-4],ax     ;add LSW of j to LSW of i
adc [bp-2],dx     ;add MSW of j and Carry to LSW of i
```

Compare this with the code for the same operation as generated on a VAX 11/780:

```
addl2 -8(fp),-4(fp)    ;add j to i
```

The 8088 code requires far more instructions to perform this operation. Operations such as long integer multiplication are even more expensive.

Similarly, dereferencing a pointer to fetch or store a value is also a complicated operation on an 8088. Consider the code generated by the assignment **ip* += **jp* where *ip* and *jp* are declared locally as pointers to long integers. For the 8088, the following code would be generated:

```
les di,[bp-2]     ;load MSW of jp into ES and LSW into DI
es:               ;use Extra Segment for next instruction
les ax,[di]       ;load MSW of *jp in ES and LSW into AX
mov dx,es         ;move MSW of *jp into DX so DX:AX == *jp
les di,[bp-4]     ;load MSW of ip into ES and LSW into DI
es:               ;use Extra Segment for next instruction
add [di],ax       ;add LSW of *jp to *ip
es:               ;use Extra Segment for next instruction
adc [di+2],dx     ;add MSW of *jp and Carry to *ip
```

Compare this with the code for the same operation as generated on a VAX 11/780:

```
addl2 *-8(fp),*-4(fp)    ;add *jp to *ip
```

Pointer comparisons are even more expensive than dereferencing since code must be generated to normalize each pointer on the 8088. This code would consist of at least a function call to a normalization routine, taking 3 or 4 bytes for each pointer variable reference.

If we were to attempt to compile the same code with one of the better C compilers available for the IBM PC today, such as the Borland International Turbo C compiler, or the Microsoft C compiler, smaller code would probably result. These newer compilers can be told to keep pointers normalized at all times. Comparisons would thus be faster, and assignments to long integer variables would result in values that could be correctly compared. They are also better at eliminating redundant reloads of segment registers, thus reducing the size and execution time of pointer operations. However, the problem of performing 32 bit arithmetic still exists. If we were to attempt to run the code on the now common 80286 based personal computers, it would be possible to bypass the operating system (since PC DOS runs in the 80286's *real* mode, which effectively limits addressing to one megabyte) and access that processor's entire 16 megabyte address space, making the size of the code image less of a problem.

There are many things that could be done to force Maple to run on an IBM compatible computer, but none of these would contribute to its general portability. Rather than expend an incredible amount of effort just to port to one class of machine, a more general solution might be desirable. Either a PC specific solution or a more general solution would require a great deal of effort, so the general solution is desirable in the long run.

The Maple Machine

This chapter discusses the history of the Maple Machine, and examines the goals which directed its development. The Machine architecture and the design decisions involved are then described in detail.

4.1. History

After attempting to port Maple to the IBM Personal Computer in August 1986 by massaging the source code and compiling it with a commercially available C compiler, and meeting with no success, it was decided that a different approach was needed. The original idea conceived by the author was to write a program that would emulate a specific larger computer (most likely the VAX 11/780), so that the executable Maple kernel could actually be ported. This idea was quickly discarded as it would be far too inefficient. Professor Gaston Gonnet of the Symbolic Computation Group at the University of Waterloo suggested the development of a higher level architecture, which could be implemented in software with some hope of reasonable performance. In other words, a program would be written that would emulate a computer architecture especially suited to running Maple.

This idea initially met with opposition, due to the potential enormity of the task involved, but the author managed to make a case for trying it. Several factors influenced the decision.

The primary objections to the project were that it would be too slow, and that it would take far too much time to write a C compiler that would generate code for the emulated architecture.

The first objection was overcome by suggesting that the emulator did not have to emulate a machine with the traditional architecture and relatively low level instruction set. The machine could be tailored to the C language, providing simple yet easily applicable primitives that correspond to C language features, thereby increasing performance over a straight-forward register based machine emulation.

The difficulty of writing a C compiler was minimized by the availability of a public domain implementation of C known as Small-C. This was a compiler for Intel 8080 based machines, written by Ron Cain, and first appeared in the May 1980 issue of Dr. Dobbs Journal of Computer Calisthenics and Orthodontia. This was enhanced by James Hendrix and republished in the December 1982 issue, and in a book devoted to the compiler [8].

4.2. Goals of the Maple Machine

There were several design goals for the development of the Maple Machine, which had a profound influence on its architecture.

First and foremost, the machine had to be suitable for running Maple on, assuming the existence of a compiler for it. This called for a 32 bit word size, and a single large linear address space, as Maple assumes both in many places in its source code.

The next most important requirement was that both the Machine itself and the code that it runs be compact. By keeping the Machine small, it would remain easy to port. By keeping both the Machine and the code that runs on it small, it would be possible to use it to run Maple on the IBM Personal Computer in the 640 kilobytes of space available. These goals required a relatively small instruction set, as well as a compact encoding of these instructions.

Another important requirement was fast execution speed. Maple is a relatively fast symbolic computation system, so some loss of performance could be tolerated. However, the final Maple Machine would be running on an IBM PC, which is at least an order of magnitude slower than a VAX 11/780. Thus, any performance loss due to interpretation would be magnified by the performance loss due to the slower hardware.

Finally, the Maple Machine had to be easy to generate good code for. The compiler available did not perform any global optimizations, and would have been difficult to modify to do so. Samples of the 8080 code generated by the compiler indicated that it was not a very good compiler, due in part to the difficulty in generating good code for the 8-bit 8080 chip. The instruction set of the Maple Machine would have to be geared towards what the compiler could generate efficiently.

4.3. Design

With these goals in mind, the process of designing the Machine's architecture was begun.

Early discussions centered on the idea of a machine that would execute Polish prefix notation. This is basically a functional notation, in which each operator precedes its operands. For example, one way of writing a prefix notation for the assignment $x = y + 2 * z$ would be:

assign(x, sum(y, product(2, z)))

Such a notation, when expressed in a well designed machine readable form, could be very compact. It would also be fairly straightforward to have a C compiler generate code in this notation, since it essentially represents a pre-order traversal of the parse tree built up during the compilation process. However, execution of such code would require a recursive interpreter. Such an interpreter would call a procedure to execute the first operator, which would then call procedures to evaluate the operands, and so on. The stack of such a machine would be implicit in the run-time stack of the emulation program, making implementation fairly simple, but performance would be poor due to the excessive number of function calls involved.

After eliminating a Polish prefix architecture, a Polish postfix architecture was considered. This appeared to be more promising, since the instructions in such an architecture could be executed by a simple loop and a stack data structure.

An article by Daniel Miller in the April 1987 issue of Byte Magazine [13] discussed the suitability of the Forth language as the output of a C compiler. Forth is a low level language originally designed by astronomer Charles Moore to control telescopes [14]. Forth appears to the programmer as a sort of Polish postfix assembly language for a stack based machine. Miller's article showed that this type of code was easily generated by a recursive descent compiler such as that for Small-C.

Other examples of stack based machine architectures also indicated positive results in terms of simplicity of implementation and compiler design. The Pascal-P [15] and Pascal-S [23] systems designed at ETH Zürich both used similar stack based architectures that were emulated by an interpreter program. Niklaus Wirth's Lilith machine [17,25] was a hardware implementation of a stack based architecture designed specifically to run Modula-2 programs [24].

Tannenbaum [20] showed that a stack machine was very well suited to structured programming languages such as Algol, Pascal, and BCPL. Code size for such a machine was typically one third of that for the same program compiled for a conventional machine. In addition, generating code for such a machine was fairly straight-forward.

Since a stack architecture would result in a smaller code image size than a conventional register based architecture, fewer instructions would have to be fetched, and therefore less time would be spent in the instruction fetching code of the emulator. As a result, execution would be faster.

With all this evidence pointing to the suitability of a stack machine both for ease of compilation and reduced code size, it was decided that this was the ideal approach.

4.3.1. The Programmer's Model

Usually, when describing the architecture of a real computer, a diagram showing the internal architecture is presented. This diagram is an abstraction that represents the machine as an assembly language programmer would perceive it. Such a diagram and the accompanying text describes the *programmer's model*. We will describe the architecture of the Maple Machine the same way, except that there is no real machine on which the diagram is based. The abstraction is implemented directly as a computer program. This of course does not preclude a hardware implementation of the Maple Machine.

Figure 1 shows the basic structure of the Maple Machine. The left side of the diagram represents the Machine's addressable memory, while the right side shows all the internal registers and the subroutine call return stack.

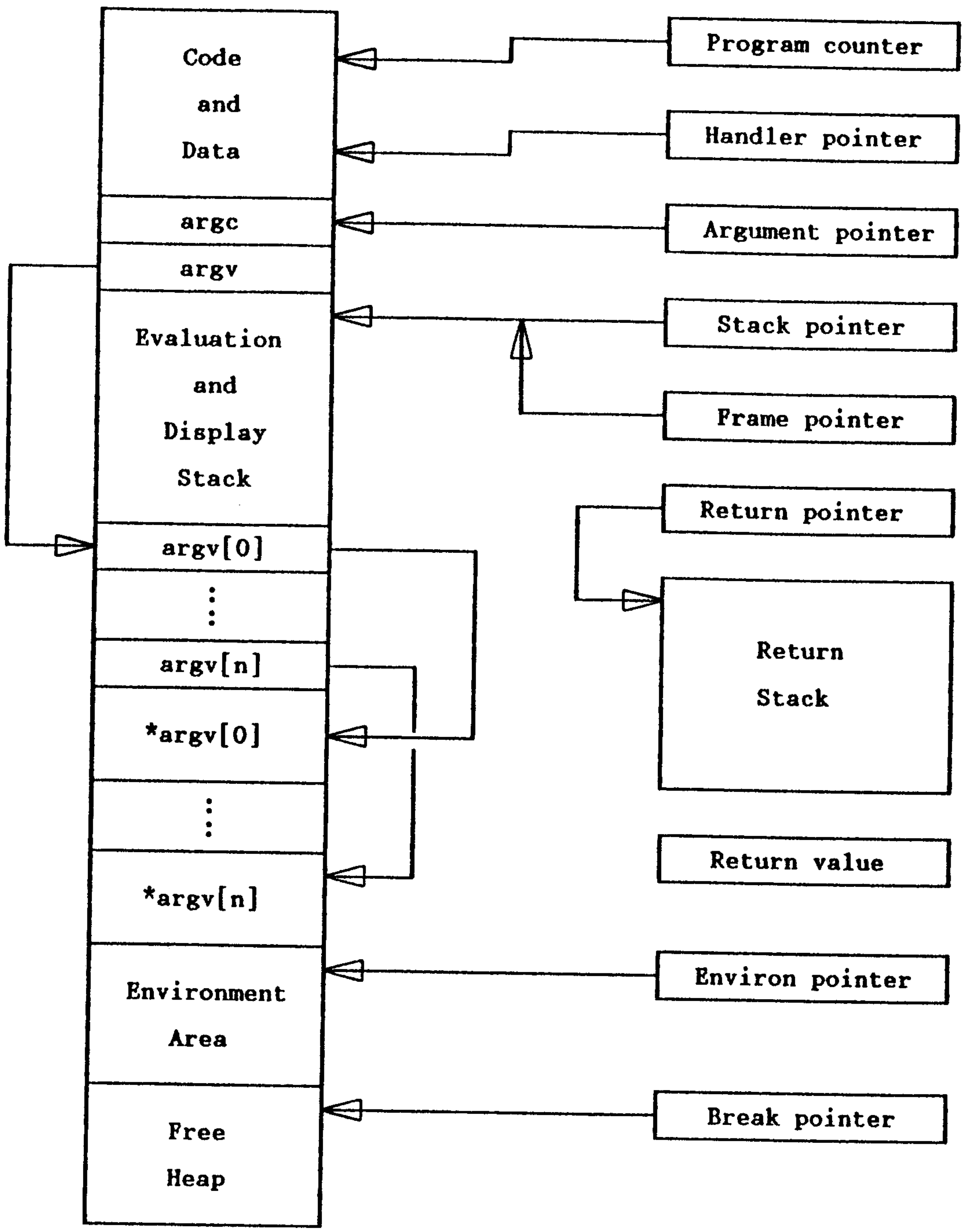

Figure 1. The architecture of the Maple Machine.

The main memory of the Maple Machine is simply a contiguous set of addressable bytes. Each byte contains 8 bits. The first byte has address zero, and each subsequent byte has an address one higher than its predecessor. Unlike most real machines where such details are a feature of the operating system, the Maple Machine imposes a predefined order on the things that are loaded into memory. There are also several pieces of data that are put into the memory by the Machine before program execution commences to allow the program to receive information from the environment in which the Maple Machine is running. These are described in more detail later.

The instructions and static data of the program that is to run on the Maple Machine are loaded into memory starting at address zero. Following this are the two parameters to the program's *main* procedure, which are discussed in detail in Chapter 6.

Following these is the evaluation stack, which is used to hold temporary results, procedure arguments, and local variables. After the stack is more parameter information for the *main* procedure.

The environment area is a "scratch-pad" in which some of the built-in functions store data so that a pointer to this data can be passed back to the executing program.

Finally, the heap consists of all the remaining memory. This can be allocated by the Maple Machine when the program requests it.

A program counter points to the byte containing the next instruction to be executed. When a program is first loaded into the Machine, the program counter is set to point at the program's entry point. This is defined by the global symbol *main* in the assembly language code of a Maple Machine program. The Maple Machine is a load-and-go machine. There is no operating system or monitor program; a program is loaded and execution begins when the Maple Machine is started. All development software for the Machine must run on either the underlying real machine, or some other real machine.

The handler pointer points to the instructions to be executed if the program running on the Maple Machine is interrupted by the user. There is a built-in function in the Machine's instruction set to set this pointer. If the user interrupts the Machine itself before this pointer has been set, then a default handler stops the Machine and control returns to the host environment.

The argument pointer points to the location in the stack of the first argument passed to the currently active procedure. The next argument is found at an address 4 bytes higher than the first, and so on. This pointer is set by the Maple Machine during a procedure call.

The frame pointer points to the location in the stack of the first local variable of the currently active procedure. The next variable is found at an address either 1 byte or 4 bytes higher than the first, depending on whether the first was byte sized or word sized. This pointer is set automatically during a procedure call.

The stack pointer points to the first word-aligned address (the address is a multiple of 4) after the last local variable. Any temporary results or parameters to further procedures are stored on the stack at or beyond this location. Word alignment is used since stack accesses are quite frequent, and many real machines impose a performance penalty when fetching words from non-aligned addresses.

The return pointer points to the next available location in the return stack. The return stack is used to hold the information associated with procedure activation. This includes the current values of the program counter, and the argument and frame pointers. The return stack was kept separate from the evaluation stack because the author finds it conceptually simpler that way; one does not need to consider the size of the return information when computing argument and local variable addresses.

The environ(ment) pointer holds the address of the environment area. This address is effectively constant for the duration of the execution of a Maple Machine program, and is returned to the program executing on the Machine whenever the built-in function *getenv()* area is called.

The break pointer contains the address of the first unused byte of the Machine's memory. A program can call the built-in memory allocation function to return this address, and increment the break pointer by some amount, thus allocating some memory for the program's use.

4.3.2. The Evaluation Stack

The evaluation stack is used for two purposes: to hold operands and results of instructions, and to hold procedure parameters and local variables.

Instructions such as "+", "*", and *NOT* each pop one or two values off the evaluation stack, perform the indicated operation, and push the result back on the stack. Other instructions may not push anything back on the stack, such as "!", which pops an address and a value off the stack and then stores the value at the memory location specified by the address. Figure 2 illustrates the execution of the "+" and "!" instructions.

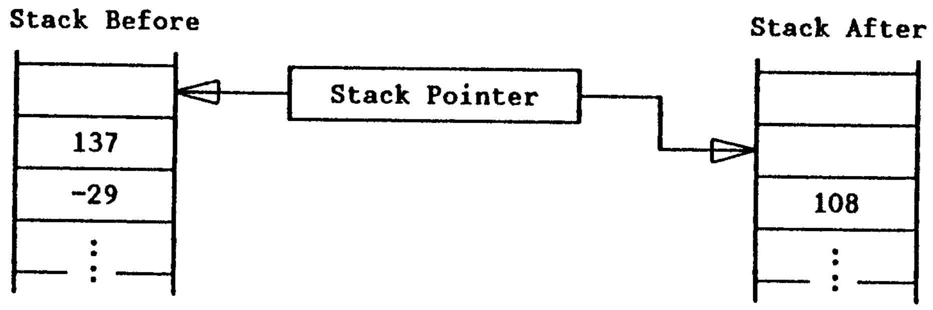

Figure 2(a). Execution of the "+" instruction.

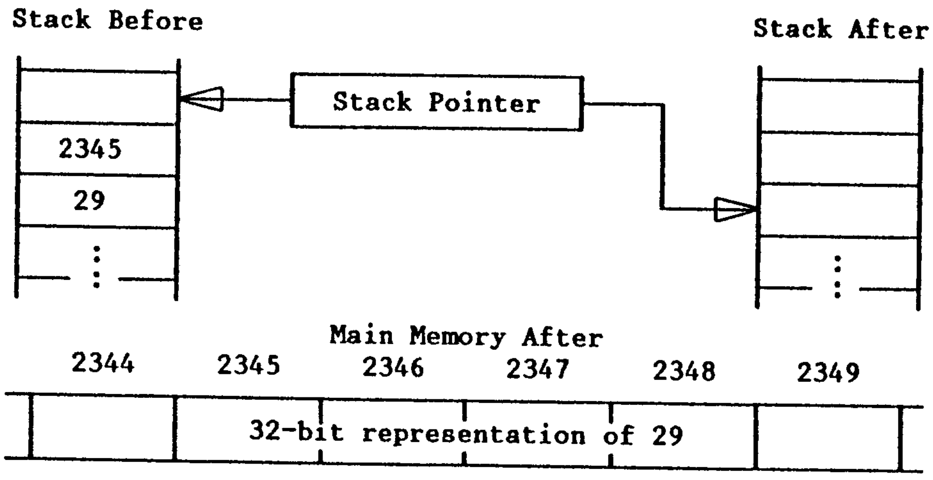

Figure 2(b). Execution of the "!" instruction.

Before a procedure is called, any parameters to that procedure are pushed on the evaluation stack. If the call is an indirect call, the address of the procedure to be called is pushed next. Finally, a count of the number of parameters is pushed. When the procedure call instruction is executed, the count (and the address if the call is indirect) is popped and control transfers to the called procedure, which may then allocate further space for its own variables. The details of these operations, complete with the effect they have on the various pointer registers, are described in the next section.

Since the evaluation stack holds local variables, and the Maple Machine is designed to run C, which allows one to compute the address of any variable, the evaluation stack (unlike the return stack) is allocated in the Maple Machine's address space.

4.3.3. The Return Stack and Procedure Calls

The return stack is used to store the current values of the program counter, frame pointer, and argument pointer registers when a procedure call is made. These registers are restored when the called procedure terminates with a *RETURN* or *RETVAL* instruction. Notice that the stack pointer register is not saved. When a procedure terminates, the stack pointer is set to the value of that procedure's argument pointer, effectively unstacking all local variables and actual parameters. This obviates the need for separate instructions to adjust the stack after a call returns, thus reducing execution time overhead. Figure 3 illustrates the procedure calling mechanism.

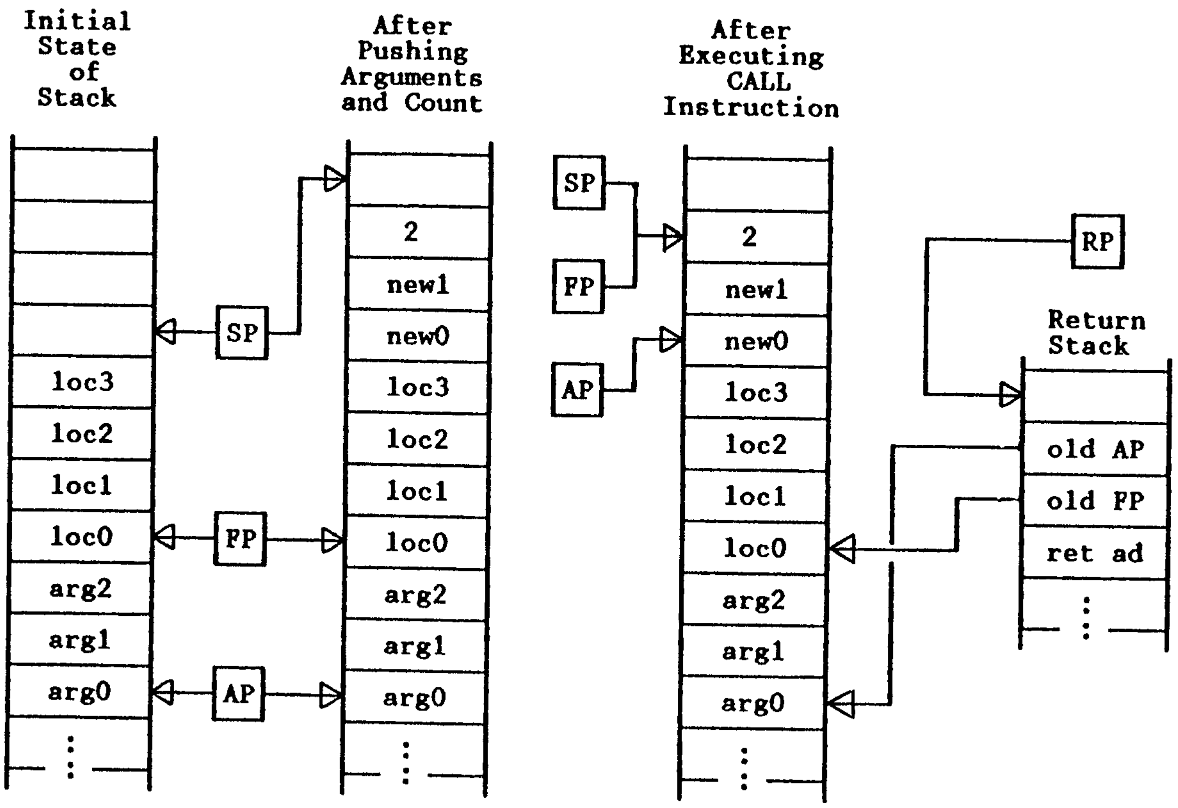

Figure 3. Procedure calling sequence.

The return stack is only accessed by the internal workings of the Maple Machine, and not by any program running on the Machine. The mechanics of procedure calls are not standard from one real machine to the next, so no portable C program would ever try to access values such as the return address or stacked frame pointer. Therefore, the return stack need not occupy the Maple Machine's address space.

4.3.4. Instruction Set

Instructions in the Maple Machine's instruction set can be broken into several categories:

1. Constants
2. Arithmetic
3. Bitwise Logical
4. Boolean
5. Comparison
6. Stack Operations
7. Memory Access
8. Control Flow
9. Hybrids

Each category is discussed separately below. The notation used to describe each instruction is a follows:

```
OPCODE              STACK-BEFORE              STACK-AFTER
```

For each of the stack pictures, only the elements of the stack relevant to the instruction are shown. The top of the stack is on the left. Symbols such as *X*, *Y*, and *Z* represent values on the stack. A string of the form *[X]* represents the value at memory location *X*. The symbol "~" indicates that there is currently nothing on the stack that will be used or has been used by the instruction.

There are also several instructions that cannot be written in the Maple Machine's assembly language. Instead, these instructions are generated by the optimizer at assembly time. Most of these are single instructions that perform the work of a sequence of two or more other instructions. These are described in detail in Chapter 7 when we deal with improvements to the system.

In addition to actual Maple Machine instructions, there are also several pseudo-instructions, or assembler directives. These tell the Maple Machine assembler to allocate storage, manage control structures, or declare procedures. These are described in Chapter 5 when we discuss the assembler.

Constants

The Maple Machine has several representations for constants. In the Maple Machine assembly language, all constants are written as signed decimal integers (Maple 4.2 does not make use of floating point arithmetic internally, so it was not implemented). The assembler then selects the smallest internal representation that is large enough to describe the constant. Since small constants such as 0 or 1 are very common in programs, a great space savings is realized by this approach. The semantics of a constant are as follows:

a_number	~	a_number

Arithmetic

Arithmetic instructions can be divided into two further categories: unary operations and binary operations. Unary operations pop one value from the stack and push one value, while binary operations pop two and push one. The standard symbols of the C programming language will be used in describing the results on the stack below.

+	X Y	X+Y
-	X Y	X-Y
*	X Y	X*Y
/	X Y	X/Y
%	X Y	X%Y
NEG	X	-X
++	X	X+1
--	X	X-1
W+	X	X+4
W-	X	X-4
SCALE+	X Y	X+4*Y

Most of these are self explanatory, but the last three deserve some clarification. In C, when the "++" or "--" operators are applied to a pointer variable, that variable is incremented or decremented by the size of the object to which it points. The two most common types of objects to which a pointer would point are characters and integers. Since integers are 32 bits, which corresponds to 4 bytes, the operations of incrementing and decrementing by 4 are quite common. The *W+* and *W-* instructions perform this using only a single instruction, instead of the equivalent sequences "4 +" and "4 -". The *SCALE+* instruction is used in array indexing. If *i* is an array of integers, then the expression *i[j]* would refer to the integer that begins *4j* bytes (*j* words) past the beginning of the array. Without the existence of the *SCALE+* instruction, the generated code to fetch this integer would be,

```
i ? j ? 4 * + ?
```

where "?" is a fetch instruction. Using *SCALE+*, this is reduced to,

```
i ? j ? SCALE+ ?
```

a savings of two instructions, and hence the overhead involved in two instruction fetches.

Bitwise

Bitwise instructions can also be grouped into unary and binary categories, according to the number of operands they pop. The bitwise instructions are:

&	X Y	X&Y
|	X Y	X|Y
^	X Y	X^Y
<<	X Y	X<<Y
>>	X Y	X>>Y
~	X	~X

Boolean

The Boolean instructions deal with truth values. Truth is represented by any non-zero integer, while falsehood is represented by zero. The result of any Boolean instruction is always either 1 (true) or 0 (false). The Boolean instructions are:

BNOT	X	!X
BOOL	X	!!X

As with almost all real machines, there are no instructions for Boolean *AND* or *OR* (corresponding to "&&" and "||" in C), because these are evaluated by conditional branching, thus skipping any further evaluation when the result of an expression becomes known. For example, the C expression,

```
a && b || c
```

would compile into the following Maple Machine code:

```
a ?
DUP
IF
    POP
    b ?
ENDIF
DUP
IFZ
    POP
    c ?
ENDIF
```

Comparison

Comparison instructions take two values from the stack, and return a Boolean value (1 or 0) depending on whether or not the comparison holds. The comparison instructions are:

<	X Y	X<Y
>	X Y	X>Y
<=	X Y	X<=Y
>=	X Y	X>=Y
==	X Y	X==Y
!=	X Y	X!=Y
U<	X Y	X<Y
U>	X Y	X>Y
U<=	X Y	X<=Y
U>=	X Y	X>=Y

The last four operators differ from the first four only in that the comparisons are made with the assumption that *X* and *Y* are unsigned quantities. The Maple Machine's C compiler does not support the **unsigned** data type, but unsigned comparisons are generated for pointer values. In practice, signed comparisons would suffice so long as the size of the address space does not exceed the largest positive signed number. The port of the Maple Machine to the Oberon system, for example, did not differentiate between signed and unsigned comparisons. Future Maple Machine implementations may eliminate the unsigned comparison operators.

Stack Operations

These instructions are used to manipulate the contents of the stack. They are often generated by the compiler in order to prepare the stack for a sequence of other operations. The stack instructions are:

DUP	X	X X
POP	X	~
ROT	X Y	Y X X
SWAP	X Y	Y X
STACK	X	N1 N2 ... NX

Forth programmers will recognize most of these by name, although the *ROT* instruction does not do the same thing as the Forth *ROT* instruction, instead it performs the equivalent of *DUP* followed by the Forth *ROT* instruction.

The *STACK* instruction pops a value off the stack, and then pushes that many undefined words back onto the stack. In actual fact, the stack pointer is simply adjusted as if that many words had been pushed. This is used in procedures to allocate space for local variables. *STACK* can deallocate space by supplying it with a negative operand, although in practice this feature is rarely used since a *RETURN* or *RETVAL* instruction will automatically deallocate any space that the procedure allocated.

Memory Access

The memory access instructions are used to fetch the values of variables from memory, and to store values to variables in memory. The fetch instructions are:

?	X	[X]
B?	X	[X]

These two instructions expect an absolute memory address on the stack, and return the word (?) or byte (*B*?) stored at that address. When a byte is fetched, it is converted to a word by setting the three most significant bytes of the word to zero. Values that have been pushed on the stack (as opposed to allocated in the stack space and then stored there) are always whole words.

For each fetch instruction, there is a corresponding store instruction that has the opposite effect. The store instructions are:

!	X Y	~
B!	X Y	~

These instructions store the value *X* into the word (!) or byte (*B*!) whose address starts at *Y*. In the case of *B!*, only the least significant byte of *X* is stored.

There are two more instructions which are not really memory access instructions, but are used in computing addresses for use by memory access instructions. These are:

$+	X	FP+X
#+	X	AP+X

These compute an address by adding an offset *X* to the current value of the frame pointer or argument pointer registers. They are used for computing the addresses of local variables and procedure parameters respectively.

Control Flow

Several instructions are used to manage the flow of control. When examining a Maple Machine assembly language program, one might encounter other instructions that appear to be control flow instructions, but these are merely assembler directives that may generate one of the instructions described in this section. The assembler directives are discussed in detail in a later section.

The following instructions are associated with procedure calling and return:

CALL	X Y	~
RETURN	~	~
RETVAL	X	~
VAL?	~	X

The *CALL* instruction is used to perform an indirect procedure call. The program first pushes the address of the procedure to call on the stack, followed by a count of the number of arguments that were pushed before that. In the description above, a call would be made to the procedure at address *Y*, with *X* arguments.

There is also another form of the call instruction that is generated by the assembler when a procedure identifier is encountered in the source text. This instruction has the calling address coded into the four bytes following the instruction, thus obviating the need to first push the address and then pop it. This instruction is generated for normal procedure calls, while the *CALL* instruction described above is generated for indirect procedure calls, such as is performed through a C pointer to a function. The direct (normal) procedure call instruction has no mnemonic since it is never written explicitly. This was inherited from Forth, where a procedure is a sequence of words, some of which are built-in primitives, while others call user defined procedures.

When a *CALL* or direct procedure call instruction is executed, the address to which execution should eventually return is pushed onto the return stack. For a *CALL*, this is the current value of the program counter (which has already been incremented by the instruction fetch code). For a direct procedure call, this is 4 more than the program counter, since the address of the called procedure follows the instruction. After this, the values of the frame and argument pointers are pushed onto the return stack. Then, the count of actual arguments is popped off of the evaluation stack and the frame pointer is set equal to the stack pointer. The argument count is then subtracted from the frame pointer to yield a new value for the argument pointer.

When a *RETURN* instruction is executed, the stack pointer is set equal to the argument pointer, effectively popping all the arguments and local variables off of the evaluation stack. Then, the argument pointer and frame pointer are popped, restoring the values they had before the procedure call. Finally, the program counter is popped to return control to the instruction following the *CALL* or direct procedure call instruction.

The *RETVAL* instruction is similar to *RETURN*, except that it first pops a 32-bit value off of the evaluation stack and stores it in the return value register, where the calling procedure can retrieve it using a *VAL?* instruction.

The Maple Machine provides several instructions to transfer the flow of control from one part of a program to another. These instructions are:

GOTO	~	~
IF	X	~
IFZ	X	~
WHILEFOR	X	~
SWITCH	X	~
CASE	~	~
DEFAULT	~	~

The *GOTO* instruction is followed in memory by a word giving the address of the next instruction to execute. This word is fetched and loaded into the program counter. The value of this word is the address of a *LABEL* directive, and is filled in by the assembler at assembly time. The instruction has no effect on either of the stacks.

The *IF* and *IFZ* instructions perform a conditional branch. A value is popped from the evaluation stack, and the branch takes place if the value is zero in the case of the *IF* instruction, or non-zero in the case of the *IFZ* instruction. In other words, a zero value causes the instructions following an *IF* to be skipped, while a non-zero value causes the instructions following an *IFZ* to be skipped. *IF* and *IFZ* instructions are followed in memory by a word giving the address of the instruction to jump to in the case of a zero or non-zero value respectively. This is the address of the instruction following the corresponding *ELSE* or *ENDIF* directive and is filled in by the assembler at assembly time.

The *WHILEFOR* instruction performs a two-way conditional branch. A value is popped from the evaluation stack, and if it is non-zero, the word immediately following the instruction is loaded into the program counter. If the value is zero, the second word following the instruction is loaded into the program counter instead. The values of these words are the addresses of the instructions following the corresponding *BEGINFOR* and *ENDFOR* directives respectively, and are filled in by the assembler at assembly time.

SWITCH performs a multi-way branch analogous to the C **switch** statement, which occurs frequently in the Maple kernel source code. The *CASE* and *DEFAULT* instructions are effectively labels to which to branch to, depending on the value popped off of the stack. The best way to describe the usage of the *SWITCH* instruction is to give an example. Consider the following piece of C code, where *a* and *b* are global integer variables:

```
switch(a) {
     case 1:
          b = 1;
          break;
     case 2:
     case 3:
          b = 2;
          break;
     default:
          b = 3;
}
```

This would compile into the following Maple Machine assembly language code:

```
a ?
SWITCH
     CASE 1
          1 DUP
          b !
          POP
          BREAK
     CASE 2
     CASE 3
          2 DUP
          b !
          POP
          BREAK
     DEFAULT
          3 DUP
          b !
          POP
ENDSWITCH
```

BREAK and *ENDSWITCH* are assembler directives discussed in a later section. The

SWITCH instruction is followed in memory by a word giving the address of the first *CASE* instruction. Each *CASE* instruction is followed by a word giving the address of the next *CASE* instruction. The very last *CASE* instruction in the sequence is followed by a word giving the address of the *DEFAULT* instruction, if any (even if the *DEFAULT* instruction lexically precedes the last *CASE* instruction), or the first instruction after the *ENDSWITCH* if there is no *DEFAULT*. Following the next-*CASE* pointer of each *CASE* instruction is a one to five byte value giving the constant that followed the *CASE* instruction in the assembly language source code. The structure of an assembled *SWITCH* statement is illustrated in Figure 4.

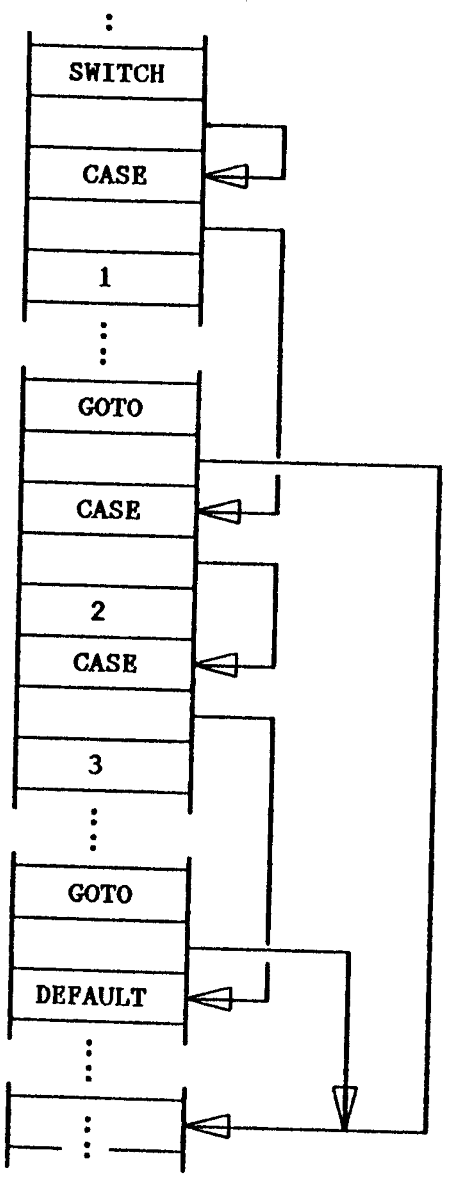

Figure 4. Structure of a SWITCH statement.

When a *SWITCH* instruction is executed, a value is popped off of the stack. The Maple Machine then traverses the chain of *CASE* instructions until it finds one that is followed by a constant matching the popped value, or it reaches the end of the *CASE* chain. Execution then continues from that point. A more efficient form of the *SWITCH* statement is discussed in section 7.2.3.

Hybrids

Since the Maple Machine was designed for running one particular C program, additional instructions could be added to perform common operations in this program.

The Maple kernel repeatedly performs several operations that are a combination of a few C operators. Normally, such a sequence of operations would be a candidate for encapsulation into a function, but for efficiency reasons, they were made into a macro instead. The sequences are quite short and the overhead of calling a function to perform them would be prohibitive. When the Maple kernel is passed through the Margay preprocessor, the macros describing these sequences are expanded into the full sequence. For our compiler to then recognize these sequences in the resulting source code would require a higher level of analysis than it currently performs. Since many macro definitions have to be changed during a port anyway, it was decided to simply undefine the macros for these operations, so that they would be passed on to the compiler unexpanded. They would then appear as function calls which the compiler could recognize and generate in-line code for instead. Effectively, expansion of these particular macros has been moved from the preprocessor to the compiler, where the compiler can better generate efficient code for them. The current Maple Machine does this with three such macros: *ID*(x), *LENGTH*(x), and *CASEID*(x). These return the identification of a Maple data structure, its length, and a special identification used in **switch** statements respectively, given a pointer to the structure. The special instructions that perform this function are:

```
?ID            X            [X]&017600000
?LENGTH        X            [X]&0177777
?CASE_ID       X            ([X]&017600000)>>16
```

Consider the following C code, with *x* declared as a pointer to an integer, and *y* declared as an integer:

```
y = ID(x);
y = LENGTH(x);
y = CASE_ID(x);
```

The following assembly language output would result if these three hybrid instructions were absent:

```
x ? ?
65535 &
y !

x ? ?
4128768 &
y !

x ? ?
4128768 &
16 >>
y !
```

With the new instructions however, the code becomes much shorter. The large masking constants do not need to be stored in the code image, and there are fewer instructions. Execution is also accelerated accordingly:

```
x ? ?LENGTH
y !

x ? ?ID
y !

x ? ?CASE_ID
y !
```

4.3.5. Intrinsic Functions

The Maple Machine has been specifically designed to be a suitable target machine for a C compiler that is compiling one particular program, namely the Maple kernel. C programs generally make use of functions from the C run-time library, a more or less standard collection of C functions to perform common actions such as file I/O, memory allocation, character classification, and more.

The obvious way to make these facilities available on the Maple Machine would of course be to write a run-time library in C, possibly based on the run-time library of an existing machine. This library could then be compiled using the Maple Machine C compiler, and linked in with the Maple kernel or other C programs to be run on the Maple Machine.

One problem with this approach however is poor performance. Library facilities such as those that do I/O perform large numbers of low level operations. If these operations were to be performed by interpreted Maple Machine instructions, such facilities would be excruciatingly slow.

Another problem is that such a library might not be portable. The code in such a library would have to access certain aspects of the underlying hardware and/or software. The premise of the Maple Machine makes this impossible, but even if it were possible, the resulting code would only work on Maple Machines running on a particular host.

Since the Machine will only be used to run C programs, and probably only to run the Maple kernel, a more suitable approach would be to provide these facilities as features of the machine itself. This is the approach that was finally taken. This makes it possible for a program running on the Machine to make use of the same facilities regardless of the underlying hardware or operating system. The same code image can be run on any Maple Machine implementation! It is up to the implementor of each Machine to ensure that the facilities provided correspond exactly to those of any other implementation.

Ideally, there would be a separate instruction corresponding to each C run-time library function that it to be implemented by the Machine. The problem with this however is that there are only 128 instruction codes available (since each instruction must fit in one byte, and the 128 codes with the most significant bit set are used for short integer constants), and 57 of these are already taken up by normal instructions. The remaining codes are reserved for use by new instructions, as is discussed in Chapter 7 in the section on optimization.

An alternate approach is to have one instruction that signals that the following byte in memory identifies one of the built in functions. This is the approach used by the Pascal-P code [15] for standard procedures such as *writeln* and *dispose*. By allocating a whole extra byte to identify the function, up to 256 functions can be built into the Maple

Machine. Fortunately, the Maple kernel only uses about 30 different library functions.

When the Maple Machine interpreter encounters this instruction, which we will call INTRINSIC, it pops the argument count from the evaluation stack, and then pops that many arguments into an internal array. It then fetches the identifying byte, and executes the code that implements the function. The code for the functions uses the arguments as they are stored in the array, and returns values using the same technique as any normal procedure call.

The Maple Machine C compiler has no knowledge of the functions built into the Maple Machine. It generates the same code for normal function calls and intrinsic function calls. It is the assembler which has a list of all the intrinsic functions, and generates an INTRINSIC instruction followed by an identification byte when it recognizes an intrinsic function.

One side effect of using intrinsic functions is that it is not possible for a C program to compute the address of one of these functions. This would only be required if some other function in the program needed to be passed the address of a function to execute, and an intrinsic function was one of these. This never happens in the Maple kernel, but if it did, it would be easily circumvented. It would only be necessary to write another function (with a different name) to encapsulate the intrinsic function, and then pass the address of that function. All other uses of the intrinsic function would still be direct.

The C run-time library functions implemented by the Maple Machine are briefly described below. They are grouped by category, and listed alphabetically within each category. For further details on the semantics of these functions, refer to the UNIX Programmer's Manual [2,22].

The most important functions implemented by the Maple Machine are the file I/O functions, since it is not possible to write these to run as Maple Machine programs. Many of these functions take a parameter indicating the file on which they are to operate. This parameter is shown as *fp* in the descriptions. The value of this parameter can be either 0, 1, 2, or a value returned by a call to *fopen()*. In the latter case, its value is meaningless to the program running on the Maple Machine. It only has meaning to the functions that require such a parameter. For example, when a Maple Machine is implemented on a 32 bit UNIX based host, the value returned by *fopen()* is a pointer to a file descriptor. This value can be represented by an integer and is returned unmodified to the program running on the Maple Machine. It is clearly meaningless to the program, since it has no way to access the real memory of the underlying hardware to which the value points. However, the other file I/O routines can use this value appropriately, usually by calling the corresponding routine in the host's C run-time library. On other Maple Machine implementations, the value returned by *fopen()* could be an index into a file table, or any other value meaningful to the underlying implementation. The following C file I/O library functions are supported:

`fclose(fp)`

The file referred to by *fp* is closed. Any output that was not yet physically written to disk will be.

`fflush(fp)`

Any buffers associated with the file referred to by *fp* are flushed, that is, physically written to disk. On some underlying machines, this function may not actually do anything.

`fopen(s,m)`

Opens the file with the name pointed to by *s* and the mode pointed to by *m*. Both the name and mode are strings in the Maple Machine's memory space, and *s* and *m* are Maple Machine memory addresses. The name should follow normal UNIX file naming conventions, and may include a path of directories. Conversion between UNIX style file names and those of the underlying operating system are done by the Maple Machine. The mode should be the string *r* or *w* for reading or writing respectively. Returns an implementation dependent value which can be used as the *fp* parameter in other file I/O calls.

`fprintf(fp,s,n1,n2,...)`

Writes the parameters *n1*, *n2*, ... to the file referred to by *fp* according to the format specified in the string pointed to by *s*. Characters in the string are written as they appear, except that any occurrence of "%s" or "%d" is replaced by a textual representation of the corresponding parameter. The sequence "%s" indicates that the parameter is to be interpreted as a pointer to a string in Maple Machine memory, while "%d" indicates that the parameter is to be interpreted as an integer. None of the modifiers or additional format specifications described in the UNIX Programmer's Manual are supported. There can be up to nine parameters after the *s* parameter. This last restriction is not exactly UNIX compatible, but is adequate for porting Maple.

`fread(p,size,n,fp)`

Reads up to *n* items (if possible) of size *size* into the Maple Machine memory pointed to by *p*, from the file referred to by *fp*. The number of items actually read is returned. If the end of the file has been reached, *fread()* returns zero.

`freopen(s,m,fp)`

Closes the file referred to by *fp*, and then opens the file whose name is pointed to in Maple Machine memory by *s*, with the mode pointed to by *m*. If successful, the function returns the value of *fp*, otherwise it returns zero.

`fwrite(p,size,n,fp)`

Writes up to *n* items (if possible) of size *size* from the Maple Machine memory pointed to by *p*, to the file referred to by *fp*. The number of items actually written is returned. If an error occurs while writing, the number of items written may be less than the number requested to be written.

`getc(fp)`

Reads one character from the file referred to by *fp* and returns it to the caller.

`printf(s,n1,n2,...)`

Does the same as *fprintf()*, except writes to the standard output. This is equivalent to fprintf(1,...);

`sprintf(t,s,n1,n2,...)`

Does the same as *fprintf()*, except that the formatted output is written into Maple Machine memory at the location pointed to by the *t* parameter, instead of to a file.

The standard C run-time library provides several functions to provide abnormal flow of control. These include functions to branch out of a block, and a function to terminate execution of the program. The corresponding functions supported by the Maple Machine are described below:

`exit(n)`

The Maple Machine, and the program running on it, terminates. If the host environment has the concept of a result code or program return value, *n* is returned as that value. Otherwise *n* is ignored.

`setjmp(p)`

The current state of the Maple Machine (stack pointer, program counter, etc.) is saved in the Maple Machine memory locations pointed to by *p*. At least five words of memory must be available for use by this function at that location. The function returns zero.

`longjmp(p,r)`

Flow of control returns to the state that was saved in the five words of Maple Machine memory pointed to by *p*. This function never returns. Instead, the program "wakes up" as if it had just returned from the *setjmp()* call that saved the state, except that the return value is now *r* instead of zero. Of course, any global variables, files, or other persistent objects that were changed since the *setjmp()* call will remain changed.

The C language does not have a character string data type. Instead, it provides arrays of characters, whose contents can be manipulated as strings by appropriate functions. Such strings are typically terminated by a 0 byte, although one could write string functions that deal with different formats of strings. The following standard string handling functions are implemented within the Maple Machine:

`strcat(s,t)`

The characters of the string in Maple Machine memory pointed to by *t*, up to and including the terminating byte, are copied to the end of the string pointed to by *s*, overwriting its terminating byte. The function returns the value *s*.

`strcmp(s,t)`

The strings pointed to by *s* and *t* are lexicographically compared. The function returns -1 if the first string precedes the second, 0 if they are equivalent, or 1 if the second precedes the first. Comparisons are done using the ASCII character set as collating sequence. Maple Machines implemented on non-ASCII hosts will make the appropriate conversions.

`strcpy(s,t)`

The characters of the string pointed to by *t*, up to and including the terminating byte, are copied to the memory location pointed to by *s*. The function returns the value of *s*.

`strlen(s)`

The number of characters starting at memory location *s*, up to but not including the terminating byte, is returned.

`strncpy(s,t,n)`

Performs the same function as *strcpy()*, except that at most *n* characters are copied. If the string at *t* contains less than *n* characters, including the terminating byte, only the string and terminating byte are copied. Otherwise, *n* bytes are copied. Large programs like Maple often make use of storage dynamically. The amount of storage needed to perform computations cannot be computed at compile time, or written into the program. The C standard library provides functions to allocate and deallocate blocks of storage. There are low level functions which simply request additional blocks from the operating system, and higher level functions to allocate and deallocate chunks of these blocks. Maple uses only one low level function, so that is the only one implemented by the Maple Machine.

`sbrk(n)`

Requests *n* bytes to be added to the memory allocated to the program so far. The address of the first byte is returned to the program. If there is not sufficient memory to satisfy the request, the function returns -1. Internally, this function has the effect of incrementing the break pointer register by *n*, and returning its old value.

UNIX based systems allow the user a great deal of control over the processes that are running. A user can interrupt a process, suspend it, place it in the background, or resume its execution. The standard library provides several functions to support these capabilities, two of which are used by the Maple kernel, and implemented by the Maple Machine. These are:

`signal(sig,handler)`

Sets up the handler function pointed to by *handler* to be executed whenever the process receives the signal specified by *sig*. The Maple Machine defines only one signal, corresponding to the UNIX signal SIG_INT. The *handler* parameter specifies the Maple Machine address of a parameterless C function. If *handler* is zero, the signal handler is disabled, and receipt of a SIG_INT signal will stop the Maple Machine and return control to the underlying operating system. On UNIX, the SIG_INT signal is generated when the user presses some key, usually BREAK or CTRL-C. In the Oberon port, the Maple Machine was set up to poll for the RETURN key to indicate a SIG_INT.

`system(s)`

The string in Maple Machine memory pointed to by *s* is passed to the shell of the underlying operating system, and executed. Clearly, the set of valid strings depends on the underlying operating system. This function is used by the Maple kernel to provide a facility for users to execute operating system commands. This command need not be implemented on Maple Machines running under operating systems that provide the capability to run multiple concurrent shells, such as Oberon. Information about the performance of a system is of interest to many users. This information allows users to tailor their programs to make the most efficient use of resources. The standard library provides several functions to provide this information, and two of these are implemented by the Maple Machine.

`time(p)`

Returns the number of seconds since 1970-Jan-01 00:00:00 GMT. If *p* is non-zero, the result is also stored in the 32 bit word in Maple Machine memory pointed to by *p*.

`times(p)`

This function does not follow the standard specified in the UNIX Programmer's Manual. It determines the amount of CPU time used by the Maple Machine since its invocation, in 60ths of a second. The result is stored in the 32 bit integer in Maple Machine memory pointed to by *p*. The equivalent UNIX function actually fills in a complex struct with various timing information. Calls to this function in an existing program will thus have to be modified. This was not considered as disadvantage for the Maple port, because the section of code that uses this function already contained several conditional compilation directives for different target machines; it was merely necessary to add a new section for the Maple Machine target. On some systems, timing information may not be available in 60ths of a second. In these cases, the Maple Machine makes use of the information that is available, and converts the result to 60ths of a second.

The few remaining functions implemented by the Maple Machine do not readily fall into any broad categories. These functions are:

`abs(n)`

Returns the absolute value of the 32 bit integer *n*.

`getenv(s)`

Calls the corresponding function in the underlying operating system to get the value of the environment variable whose name is pointed to in Maple Machine memory by *s*. The value, a string, is copied into the area of Maple Machine memory pointed to by the environment pointer register. The contents of this register are returned to the caller of the function. When the Maple Machine is implemented on a system that does not have the concept of environment variables, an empty string is copied to the environment area.

`isatty(n)`

Returns non-zero if the special file identified by *n* is associated with interactive input our output, ie. a terminal. The parameter *n* can be one of 0, 1, or 2, corresponding to the standard input, standard output, and standard error output respectively. On underlying machines which do not support the concept of I/O redirection, this function always returns non-zero.

Development Tools

5.1. The Compiler

Since the Maple kernel is written in Margay, a macro language front end to C, a C compiler was needed to compile Maple for the Maple Machine. Although modern compiler design methodologies and the nearly LALR(1) grammar of C make writing a C compiler far easier than compiler writing has ever been, it is still a decidedly non-trivial task. This was one of the major objections to beginning the Maple Machine project in the first place.

Fortunately, a compiler for a subset of the C programming language was available to the author, so that the task did not have to start from scratch.

5.1.1. The Small-C Compiler

The Maple Machine C compiler is based on Ron Cain's public domain Small-C compiler whose source code was first published in the May 1980 issue of Dr. Dobbs Journal of Computer Calisthenics and Orthodontia. An enhanced version was published by James Hendrix in the December 1982 issue, as well as in a book devoted entirely to the subject [8].

5.1.2. Shortcomings of the Compiler

The Small-C compiler only accepted a subset of the C programming language, the most notable omissions being type casts, arrays of more than one dimension, pointer indirection of more than one level, the **struct** facility, the **typedef** facility, and initialization of arrays and local variables. Fortunately, Maple does not uses **structs** or **typedefs**, as these would have been the most difficult to add.

Maple does however make extensive use of pointer indirection up to four levels deep. It also makes use of type casts, and many of its internal data structures are initialized at compile time. All of these features had to be added.

The version of the Small-C compiler that the author of this thesis had on hand was accompanied by a code generator for the Intel 8088, the processor used in the IBM PC. It was decided to retain this code generator so that the compiler could be tested after each modification by generating 8088 code and running that code on a PC.

5.1.3. Changes to the Compiler

The Small-C compiler was written by hobbyists, and was not very well constructed in some respects. It was built around a recursive decent parser, but the parser actually dealt with the source text at the character level in each procedure. There was no distinct lexical analyser which allowed the parser to deal simply with tokens. As a result, every parser procedure was littered with code to make string comparisons and skip blanks in the source being compiled. The first modification that was made was to write a simple lexical analyser to break the input into tokens, discarding irrelevant characters such as spaces and comments. The parser could then deal with these tokens, which were represented as small integers, rather than with free-form character strings. This had the effect of reducing the size and increasing the speed of the compiler, as well as making it easier to add the other new features.

Once the lexical analyser was in place, it was much simpler to work with the rest of the compiler. Most of the other new features to be added were concerned with data types. At first glance, it appeared as if it would be possible to add these in a rather straightforward manner, but the existing symbol table structure was inadequate for the task. The structure had to be modified to provide additional fields that described the attributes of a type independently of its base type. Attributes indicate whether a type is an array, a pointer, or a function, while the base type is the type of the elements of the array, target of the pointer, or value returned by the function. The original implementation did not allow for arrays of pointers, functions returning pointers, or even pointers to pointers. Modifying the symbol table was difficult because it was implemented as a single array of integers. Each entry in the table consisted of a fixed length section of this array. The fields of each entry were accessed by numeric literal constants added to an index variable, making the code very hard to follow. This implementation was not changed to use **structs** because it was deemed desirable that the compiler could compile itself. The numeric offsets were given symbolic names however.

Additional modifications were made to the symbol table structure to record initializers for global and local variables. The addition of arrays of pointers meant that the code to parse initializations was no longer adequate since one might want to initialize an array of character pointers with a list of strings. Initializers for local variable also required modifications to the parser itself, since this feature was not supported. Initializers had to be accumulated for the local variables until all local declarations for a function were processed, and then converted into code to actually perform the initializations.

Type casts were rather straightforward to add, since they do not really do anything except change the compiler's type information about the expression it is parsing.

5.1.4. Compiler Bugs

In addition to the modifications described above, several bugs were fixed. By examining the compiler's source code, it was discovered that the semantics differed from real C in some subtle ways. The most notable of these was the conditional operator (? and :). The precedence of this operator was not the same as in real C. The implementation of array element access was also non-standard. Kernighan and Ritchie [10] define the expression *e1[e2]* to be equivalent to **(e1+e2)*, where one of *e1* or *e2* is an array name or a pointer. Small-C did not handle this properly when *e2* was the array name or pointer. The Maple source does not contain any subscript expressions written in this "inside-out" way, but it was decided that if this was not done correctly, there might be other bugs lurking in the subscript expression parsing code. Therefore, it was rewritten. Another bug was discovered in the code generation for the dreaded **goto** statement. The generated code was correct so long as the **goto** did not jump out of a block containing local declarations. This was corrected by having each label directive generate code to compute the correct value of the stack pointer relative to the frame pointer. This was later made unnecessary by the improved procedure call mechanism described in Chapter 7.

5.1.5. Preliminary Testing

Once the compiler had been enhanced to accept the subset of C used by Maple, it was tested by compiling large parts of the Maple source code into 8088 assembly language. These compiled modules would obviously not run, since the compiler was generating code for a 16 bit machine with a 64K address space (by keeping all four segment registers fixed), but the exercise showed that the compiler was capable of correctly parsing the Maple source code. The generated code was examined by hand to see if it would be correct if only the 8088 were a 32 bit machine.

5.2. The Assembler/Linker

Although it would have been possible to write the compiler to generate binary object modules suitable for linking and execution on the Maple Machine, there were several reasons that an intermediate assembly language code was desired.

Assembly language in human readable form is much easier to examine for correctness than the equivalent code in a binary object file. A disassembler could have been written to turn binary object files into a human readable form, but much useful information, such as symbolic names and lines of C source text would thus be lost. If the compiler were to go to the trouble of including these in the object file, it may as well just emit assembly language.

A separate assembly language also provides an opportunity to test the Maple Machine interpreter by hand coding assembly language programs for it. Specific instructions can be tested without having to write a C program that will (hopefully) generate that instruction.

Finally, it turns out that the assembler is an ideal place in which to perform peephole optimization on the code generated by the compiler. When a compiler generates two instructions that are adjacent in the output, the sections of code that generated them might be widely separated. The assembler on the other hand deals with instructions in a very sequential manner, well suited to the examination of short instruction sequences.

5.2.1. Overview

The Maple Machine assembler is a simple program that will read one or more assembly language source files, generate instruction opcodes for each instruction mnemonic encountered, resolve references between source files, perform peephole optimization on the generated instruction sequence, and output an executable load module suitable for the Maple Machine interpreter.

The assembler performs a single pass over the source files, building up a code image in memory. Forward references are resolved by maintaining internal lists of different categories of such references, and then backpatching the code image when the referenced object is encountered. This approach results in relatively high memory requirements, but this was considered acceptable since the assembler could always be run on some suitable host. After all, the code image that is generated will be portable to any machine with a Maple Machine implementation, so the assembler will theoretically never be needed after the first successful port.

As each instruction is generated, a peephole optimizer looks at the last few instructions generated, and determines if any sequence can be replaced by a shorter sequence. If so, the appropriate changes are made and any affected forward reference lists are updated. The optimizer is completely table driven, meaning that new optimizations can be inserted by simply adding new entries to the table and recompiling the assembler. The optimizer is discussed in detail in Chapter 7.

5.2.2. Why Link at the Assembly Level?

In addition to its function as a traditional assembler, the Maple Machine assembler also performs the operations traditionally performed by a separate linker. There are several reasons for this.

The compiler generates scope information for each identifier that it generates, so that the assembler can deal appropriately with different objects of the same name. Since the assembler is already using some of this information, it seems reasonable that it can make use of all of it and perform the linking function while it is at it. The time taken to resolve references, especially with the single pass technique used, is surely less than the time it would take to write the information to a file, only to have the linker read it back.

Traditionally, the main reason to have a separate linker was speed. A linker allows one to recompile only those parts of a program that have been changed, and then just link in the object files for the other parts. When used in conjunction with a compiler that generates linkable object modules, this makes sense; recompiling all the source code for a large program is much more complicated than linking in precompiled modules. When using a linker to link modules produced by an assembler however, the time taken to link can often be just as long as the time it would take to reassemble all the source code. This is as a result of the relative simplicity of an assembler relative to a compiler. In effect, the Maple Machine assembler is a linker that takes human readable modules as input, and produces one machine readable module as output.

Informal measurements of the time required to compile the entire Maple kernel and link it into an executable load module indicate that this approach is efficient. The time required to compile Maple with the 4.3 BSD UNIX C compiler, and link it, is approximately the same as the time required to compile Maple with the Maple Machine C compiler, and assemble/link it.

5.2.3. Instruction Format

The Maple Machine assembler accepts free-form input. Since the Maple Machine is stack based, instructions do not have operands. The traditional assembler requirement of one instruction per line is therefore non-existent. There is no ambiguity about where one instruction ends and the next one begins.

Identifiers can be any sequence of non-whitespace characters that is not an instruction or assembler directive. For example, the following are all valid identifiers:

```
FOO?          B_R          12B          +-*/
```

Note that *123* is not a valid identifier, since it denotes the instruction to load the constant 123 onto the stack.

Comments, which are ignored by the assembler, are delimited by the sequence "$-$" on both ends of the comment. A single comment may extend over multiple lines.

The examples of assembly language output from the C compiler in this thesis have been edited by hand to show program structure using indentation. The compiler does not generate neatly indented code like this, but the assembler will assemble such code.

5.2.4. Assembler Directives

In addition to the instructions described in Chapter 4, of which there were 60, there are several assembler directives, or pseudo-instructions which may generate one or more actual instructions, or cause storage to be allocated.

The syntax of the assembler directives will be described using a variation of Backus-Naur notation as follows:

1. Any sequence of non-blank characters beginning with a capital letter must be typed as it appears.
2. Any sequence of characters enclosed in backquotes must be typed as it appears, but without the backquotes.
3. Lowercase symbols indicate placeholders that are replaced by appropriate values in specific instances of the construct.
4. An item or sequence of items enclosed in braces, followed by an asterisk, may appear one or more times, or not at all.

Storage allocation directives are used to set aside and possibly initialize one or more bytes or words of the Maple Machine's memory to hold simple variables or arrays. To declare a simple integer (word) or character (byte) variable with no explicit initialization, the *INT* or *CHAR* directive is used. The syntax is:

```
INT identifier
CHAR identifier
```

An array of integer or character variables is declared using the *INT[]* or *CHAR[]* directives as follows:

```
INT[] identifier size
CHAR[] identifier size
```

The value specified by *size* is the number of bytes that the variable is to occupy. Thus, an integer array with 5 elements would be declared by specifying a size of 20 bytes.

Variables can also be declared with explicit initial values. These values are written into the code image file, so no code is generated to actually perform the initializations. The syntax for declaring initialized variables is:

```
INTDEF identifier initializer
CHARDEF identifier initializer
INTDEF[] identifier size { initializer }*
CHARDEF[] identifier size { initializer }*
```

The definition of an initializer is given below. Notice that a size is specified for initialized arrays. This size is is used to determine how much space to allocate for the array. If the number of initializers given is not enough to fill that space, the remainder is filled with 0 bytes. If too many initializers are given, any beyond the allocated space are discarded.

An initializer can either be a signed decimal integer, or a text string with the following syntax:

```
TEXT: `"` characters `"`
```

Text string initializers do not represent the characters in them, but rather the address of the text string in memory. Thus, text string initializers only make sense for *INTDEF* and *INTDEF[]* declarations, where the address of the string will become the initial value of the variable.

Sometimes, it is desirable to declare global variables with a limited scope. Variables declared using *INT*, *CHAR*, or one of the other storage allocation directives will normally be visible throughout source file being assembled. A *STATIC* declaration, of the form,

```
STATIC identifier
```

will terminate the scope of a variable. The variable will not be visible past the point of the *STATIC* declaration. In fact, a new variable with the same identifier can be declared without danger of conflict with the earlier variable.

The special identifier *TEXT?*, when declared by a *CHARDEF[]* directive, is automatically made static at that point. All earlier unresolved references to it are filled in. Further references to *TEXT?* will refer to the next declaration of *TEXT?*. This identifier is declared by the C compiler to refer to all the string constants in one compilation unit. The assembly language output of each compilation unit contains its own declaration of *TEXT?*, such as in the following example:

```
CHARDEF[] TEXT? "All the strings in the preceding module..."
```

The strings are actually output by the compiler as a sequence of numeric constants, so that control characters in the string to not interfere with the assembler's parsing procedure.

Instructions in a Maple Machine program are grouped into procedures, which correspond to C functions. Each procedure is given a name, and can be called by mentioning the name in an assembly language program. A procedure is declared as follows:

```
`:` identifier
```

All instructions following this declaration up to the assembler directive,

```
`;`
```

then become part of the procedure. Parameters to procedures are not declared at the assembly language level; they are simply accessed relative to the argument pointer using instructions such as "#+".

Within a procedure, the *GOTO* instruction can be used to branch to an address. The *LABEL* directive is used to give symbolic names to such addresses, and has the following syntax:

```
LABEL identifier
```

A given label can only be declared once in a source file. Since multiple C compiler output files are concatenated and fed to the assembler as a single source file, the compiler ensures that any labels generated are unique by prepending each occurrence of a label with the current function name and an underscore character.

The primary difference between functions and labels is that the assembler keeps track of them separately, and recognizes a function invocation when it sees one.

There are several assembler directives to aid the C compiler in generating code for the **for** control structure. For example, consider the following piece of C code, where *i* and *j* are declared globally as integers:

```
for(i = 0; i < 10; ++i)
        j *= 2;
```

This C code would result in the following assembly language output:

```
        0
        DUP
        i !
        POP
FOR
        i ?
        10 >
WHILEFOR
        i ? ++
        DUP
        i !
        POP
BEGINFOR
        j ?
        2 *
        DUP
        j !
        POP
ENDFOR
```

WHILEFOR is a Maple Machine instruction, but *FOR*, *BEGINFOR*, and *ENDFOR* are all assembler directives. The *FOR* directive creates a descriptor that is used by the assembler to keep track of the various parts of the *FOR* control structure. These descriptors are stored on a stack, so that control structures can be arbitrarily nested.

The descriptor is initially filled in with only the address at which the *FOR* directive occurred. When the assembler encounters the *WHILEFOR* instruction, its location is noted in the structure descriptor, so that *WHILEFOR*'s two parameters can be filled in later.

When the *BEGINFOR* directive is encountered, the assembler emits a *GOTO* instruction with the address of the *FOR* directive as its parameter. It also fills in one of the address fields of the *WHILEFOR* instruction with the address of the instruction following the *BEGINFOR* directive. On encountering the *ENDFOR*, another *GOTO* is emitted with the address of the first instruction after the *WHILEFOR* as its parameter. The other address field of the *WHILEFOR* instruction is then filled in with the address of the first instruction after the *ENDFOR*. The links in the resulting code image are illustrated in Figure 5.

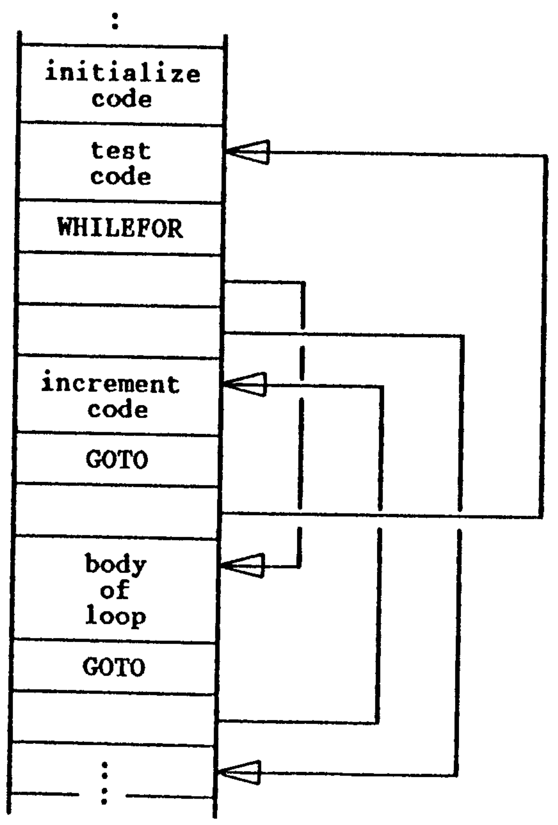

Figure 5. Structure of a FOR statement.

The reason for the existence of the *GOTO* generating directives is to simplify the compiler, and to allow it to be debugged more easily. The directives result in assembly language code whose structure is easily discerned, and is thus easily examined for compilation errors.

In addition to the **for** style of loop, C also supports two simpler loop styles: the **while** loop, and the **do** loop. Consider the following two pieces of C code:

```
while(i < 10)                  do
      ++i;                           ++i;
                               while(i < 10);
```

The compiler will produce the following assembly language outputs for these two code samples:

```
LOOP                           LOOP
      i ?                            i ?
      10 >                           ++
WHILE                                DUP
      i ?                            i !
      ++                             POP
      DUP                            i ?
      i !                            10 >
      POP                      WHILE
ENDLOOP                        ENDLOOP
```

When the assembler encounters the *LOOP* directive, it allocates a control structure descriptor and fills in the address of the instruction following the directive. When the *WHILE* directive is encountered, an *IF* instruction is emitted, and its location is stored in the descriptor. Finally, on encountering the *ENDLOOP* directive, a *GOTO* instruction is emitted, followed by the address of the first instruction after the *LOOP* directive. The address field of the *IF* instruction is then filled in to point at the first instruction after this *GOTO* instruction. Thus, the same assembler control structures are used for both the **while** and **do** control structures in C.

C supports two different selection control structures. An **if** statement is used to conditionally execute a section of code depending on the result of some expression. A **switch** statement is used to select one of several sections of code to execute. Consider the two **if** statements below, one with an **else** part and one without:

```
if(i < 10)                          if(i < 10)
        ++i;                                ++i;
                                    else
                                            --i;
```

The compiler would generate the following code for these two statements:

```
        i ?                                 i ?
        10 >                                10 >
IF                                  IF
        i ?                                 i ?
        ++                                  ++
        DUP                                 DUP
        i !                                 i !
        POP                                 POP
ENDIF                               ELSE
                                            i ?
                                            --
                                            DUP
                                            i !
                                            POP
                                    ENDIF
```

When the *IF* instruction is encountered, it is emitted, and a control structure descriptor is allocated and filled in with the location of the *IF* instruction.

When an *ELSE* directive is encountered, a *GOTO* instruction is emitted and its location is stored in the structure descriptor. The address parameter of the previous *IF* instruction is then filled in with the address of the next instruction following this *GOTO* instruction.

On encountering an *ENDIF* directive, the assembler looks at the structure descriptor to see if there was an *ELSE* part corresponding to the last *IF*. If there was an *ELSE* part, the parameter of the *GOTO* instruction emitted for the *ELSE* directive is filled in with the address of the next instruction after the *ENDIF* directive. If there was no *ELSE* part, then the parameter of the *IF* is filled in with this address instead. The resulting links are illustrated in Figure 6.

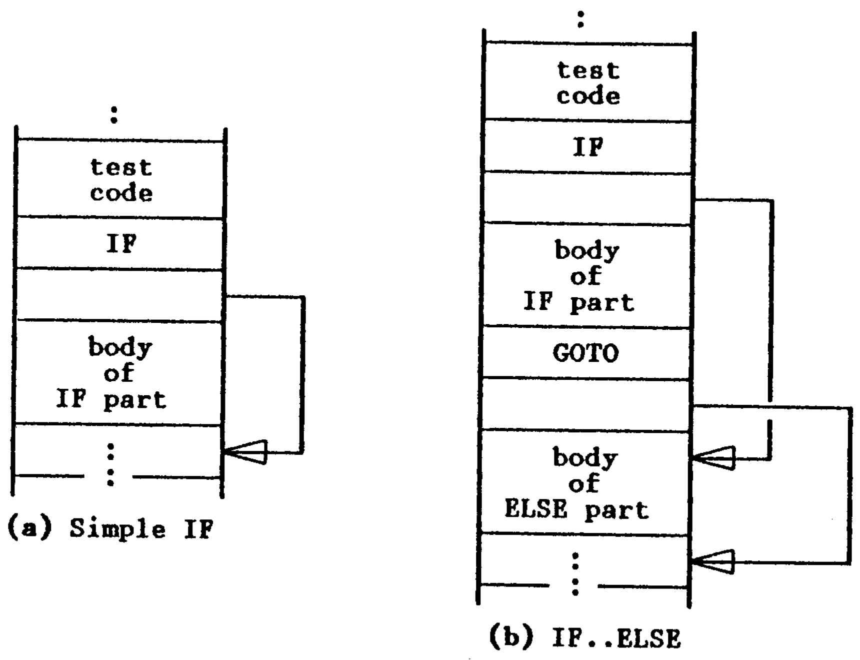

Figure 6. Structure of IF statements.

The C **switch** statement compiles into a *SWITCH* instruction, one or more *CASE* instructions, possibly a *DEFAULT* instruction, and an *ENDSWITCH* directive. When the assembler encounters the *ENDSWITCH* directive, it fills in any remaining unresolved references in the *SWITCH*, *CASE*, and *DEFAULT* instructions as was described in Chapter 4.

C provides two statements that allow abnormal control flow. The **break** statement allows the program to terminate the current loop or **switch** statement, even if the termination conditions (in the case of a **for** or **while** statement) have not been met. The **continue** statement transfers control directly to the test part of a **while** loop, or the increment part of a **for** loop. Two assembler directives provide this functionality.

The *BREAK* directive results in the emission of a *GOTO* instruction. The address field of the *GOTO* instruction is temporarily filled in with the address of the last *BREAK* directive, or zero if there were no earlier *BREAK* directives in the current control structure. The address of the current *BREAK* directive is then stored in the control structure descriptor maintained by the assembler. When the end of a control structure is encountered, signalled by an *ENDLOOP*, *ENDFOR*, or *ENDSWITCH* directive, the address of the next instruction following the control structure is filled into the address field of each of the *GOTO* instructions, which are found by traversing the links currently stored in these fields.

The *CONTINUE* directive is handled the same way, except that the address of the test or increment part of the current loop, as opposed to the address of the first instruction following the loop, is filled into the address fields of the *GOTO* instructions instead.

5.2.5. Files Produced

The assembler produces from one to three different output files, depending on the options specified when it is invoked.

The primary output file is of course the assembled code image. This file is simply a sequence of bytes, ready to be loaded into the Maple Machine's memory beginning at address zero. The last four bytes of the file are interpreted as a word that gives the address of the first instruction to be executed. This word is the address of a procedure called *main* if there is one, or zero otherwise.

If the mapping option is enabled, the assembler also produces a map file. The map file lists each identifier defined in the program, and the type and address of that identifier. An identifier can be of one of four types: *CHAR*, *INT*, *LABEL*, or *FUNCTION*. An *INT* or *CHAR* identifier can refer to either a single word or byte variable, or an array of such. A label is simply a point in memory used as the target of a conditional or unconditional branch; it does not actually occupy any memory. A *FUNCTION* identifier is one that was defined using the ":" directive.

If the correlation option is enabled, the assembler produces a correlation file. This file indicates the correspondence between assembly language source code lines and instruction addresses. This is used when debugging to find the source line that resulted in the instruction at a given address.

The First Implementation

In this chapter, we describe the process of creating the first working Maple Machine implementation. Like most large projects, many problems and obstacles made the process much more interesting.

6.1. Overview of the Interpreter

The first Maple Machine interpreter was written in the C programming language to run on a VAX 11/785 minicomputer under 4.3 BSD UNIX. The only goal was to make it work; efficiency would be considered later.

The interpreter is a C program that reads a Maple Machine code image into memory, initializes various registers and pointers, and proceeds to interpret the code image. The interpreter is invoked from the UNIX shell prompt, with the file name of the code image file as the first parameter. This is followed by any parameters that are to be passed to the interpreted program.

6.2. Implementation Strategy

Since the primary goal of the first implementation attempt was simply to get something running, efficiency was not considered in the coding of the interpreter. Clarity and maintainability of the code were much more important. Thus, the interpreter was written in as clear and readable a way as the author (a code hacker at heart) was capable of.

The Maple Machine's main memory was simply declared as a large array (approximately 1 million elements) of characters. Maple Machine addresses were thus integers beginning from zero.

When the interpreter is started, the code image is read into memory and the variable used for the program counter is set to the address specified at the end of the code image file as the entry point.

Two words are filled in above the space occupied by the code, giving the number of command line parameters (not counting the name of the interpreter itself), and the address of where these parameters are stored. These are normally referenced by the parameters *argc* and *argv* in the *main()* function of a C program. Space is then allocated above this for the evaluation stack. Pointers to each command line parameter are then written into the memory immediately beyond the stack, followed by the actual text of the parameters.

With all the relevant information loaded into memory, the various internal pointers, such as the stack pointer, are set to their initial values as described in Chapter 4. Control is then passed to the function responsible for program execution.

The actual interpretation is performed by a simple loop. At the top of the loop, the instruction whose address is in the program counter is fetched from the memory array, and the program counter is incremented. The instruction is then used as the selector in a **switch** statement, thus transferring control to the appropriate section of code for execution of that instruction. This is actually a rather efficient method, since the 4.3 BSD UNIX C compiler generates a VAX **case** instruction to index into a jump table and transfer control directly to the appropriate code.

Instructions which must access the evaluation stack directly do so with an expression such as,

```
evalstack[evalsp]
```

where *evalstack* is a pointer to the real machine address of the part of the memory array in which the evaluation stack resides, and *evalsp* is an integer representing the number of words currently on the stack. Note that *evalstack* is a pointer to a 32 bit integer, even though it points into the middle of an array of 8 bit characters. This allows words to be fetched from the stack using one VAX word load instruction, instead of four byte load instructions and a lot of arithmetic. Word fetches from other memory locations, such as might be generated by the Maple Machine instructions,

```
12345 ?
```

are handled in a similar way. The Maple Machine address (an integer), is added to the real machine address of the memory array, resulting in a pointer to a character. This is then type cast into a pointer to an integer, and then dereferenced. For example, the C code that executes the Maple Machine's "?" (fetch) instruction is simply:

```
evalstack[evalsp-1] =
     *(long *)(memory + evalstack[evalsp-1])
```

By writing the interpreter in such a straightforward manner, errors due to bizarre "tricks" were minimized. This would of course greatly reduce the time required to create a functioning implementation. Surprisingly, without using any tricky coding, the 4.3 BSD UNIX C compiler managed to generate very good code. This is probably due in part to the close parallel between the architecture of the VAX and the features provided by the C programming language.

6.3. Initial Debugging

Once the interpreter was written, it was desired to perform some preliminary tests in order to root out any coding errors that may have crept in during implementation. Testing it by actually attempting the Maple port would have been foolish. Therefore, several small test programs were written to test various features of the Maple Machine system. These programs were small enough so that any resulting errors would be immediately indicative of where the problem was.

6.3.1. Some Small Test Programs and Results

The first test program written for the Maple Machine was the one that every programmer writes as his first program on a new system or in a new language. This of course is the one that simply prints "Hello world!" and then terminates. The C source code for this program is:

```
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
{
     printf("Hello world!0);
}
```

There was of course no need to first write the *printf()* library function since this is built into the Maple Machine. The resulting Maple Machine assembly language program is:

```
: main
     0 TEXT? +
     1 printf
     VAL?
     POP
RETURN ;

CHARDEF [] TEXT? 14
     72 101 108 108 111 32 119 111 114 108 100 33 10 0
```

This was assembled by the assembler into a 31 byte executable file, of which the last four bytes indicated the entry point at location zero.

The simple test program immediately uncovered several errors in the Maple Machine system. Assembler errors resulted in misplaced references, and interpreter errors included improper use of the stack and frame pointers. An octal dump of the executable file quickly uncovered the assembler produced errors, while an examination of the interpreter's code in relation to the test program pointed out the problems in the interpreter.

More complex test programs, making use of loops, separate functions, arrays, and other features of C were written, and more errors were discovered. Eventually, errors stopped appearing (although the author did not take this as an indication that there were no more) and the processes of porting the Maple kernel was begun.

6.4. Getting Maple Running

The first step in actually porting the Maple kernel to the Maple Machine was to compile the source code, and assemble/link the resulting modules. After making changes to all the macro definitions so that they would be appropriate to the Maple Machine, the compilation could begin.

Each source file was passed through the Margay preprocessor, and the resulting "pure" C language output was given to the Maple Machine C compiler. Unfortunately, this process was not without problems. A few of the C source files contained constructs slightly beyond the capabilities of the compiler. The compiler was modified several times to extend its capabilities, and all the files compiled up to that point were recompiled. This task was greatly simplified by the UNIX *make* facility, which allows the programmer to specify the various files involved in a compilation, and the dependencies between them. In this particular case, the assembly language output files were made dependent not only on the input files, but also on the last revision date of the compiler. If the compiler was more recent than an output file, the corresponding input file was recompiled. Needless to say, this facility prevented a lot of potential errors that would result from modules generated by different revisions of the compiler.

Once all the parts of the Maple kernel had been compiled, the resulting assembly language files were concatenated and passed to the assembler. This of course unearthed several bugs in the assembler, as well as some basic design errors in the system as a whole. The largest such error was the realization that a large program was bound to have more than one **goto** target label with the same name, and that the assembler's treatment of labels as having global scope was not suitable. Unfortunately, the design of the assembler would have made it difficult to change this. Instead, the compiler was modified to prepend the current function name to each label reference, assuring that such labels would be unique in the assembly language output. This of course necessitated recompilation of all the source files.

The entire process of producing an executable went through several phases of compilation and assembly, until a code image file resulted without any reported errors. The problem with a multiple phase system such as the compiler-assembler-interpreter configuration discussed here is that an error occurring at any stage may be the result of a problem at that stage or any of the earlier stages. Thus, the further one gets in completing a whole cycle, the longer it takes to correct an error. Errors early in the cycle can be repaired quickly, but later errors can take a long time to find, and result in having to repeat the whole cycle up to that point.

After finally producing a code image without apparent error, there was still the possibility of a bug in the interpreter itself, as well as the compiler and assembler. The only output that resulted from the first test run was the message:

```
Segmentation fault
```

At this point, finding the cause of an error was a formidable task. Any particular error could be the result of a compiler bug, a problem with the assembler, a mistake in the interpreter, an undiscovered bug in the Maple kernel itself, or a really basic design flaw. To make error tracking easier, various debugging aids were slowly added to the interpreter.

6.4.1. Debugging Facilities

When debugging a program on a conventional computer such as a VAX or IBM Personal Computer, running a conventional operating system such as UNIX or PC-DOS, several tools are available.

A debugger is a program that is used to run the program being debugged in a controlled manner. The debugger tries not to give control of the computer to the program under test, since that program may not be working properly. Debuggers provide facilities to execute the program one machine instruction at a time, or possibly one source language statement at a time. In essence (although not usually in fact), debuggers interpret the machine language code image. They usually also provide facilities to produce a trace, or listing giving the values of variables and describing the flow of control through the program. Unfortunately, no debugger is perfect, and it is nearly always possible for the program being debugged (or the debugger itself) to crash anyway. On machines such as a VAX, which provide memory protection, the debugger can usually recover from this situation and provide the programmer with some useful information. On smaller machines such as the IBM PC however, there is usually little or nothing the debugger can do.

The first attempts at debugging the Maple Machine running the Maple kernel concentrated on using the UNIX *dbx* program. This allowed the author to monitor the execution of the interpreter. The first result of this was that the point in the interpreter in which the segmentation fault occurred was immediately pinpointed. The fault occurred when the interpreter was attempting to execute a "?" (fetch) instruction, and the value on the top of the stack was larger than the size of the Maple Machine's address space. Unfortunately, this information was not very useful because the interpreter had executed thousands of fetch instructions before the offending one, with no ill effect. The problem was obviously not with the fetch instruction, but with the means by which the incorrect value got onto the top of the stack. The *dbx* program also printed out the contents of the global variable *pc*, which the interpreter uses as a program counter, so it was possible to tell which fetch instruction in the code image was causing the problem. Again, this was not sufficient to indicate why the problem was occurring. It would be nice if it were possible to run the program backwards, closing in on the point where the error began, but the debuggers currently available do not allow this.

It became clear that debugging the Maple Machine by running *dbx* on the interpreter would never succeed, as the information provided by *dbx* was too far removed from the problem. After all, the Maple Machine interpreter was crashing because the program running on it was crashing; a debugger that ran on the Maple Machine itself would be more appropriate.

Writing a debugger to run on the Maple Machine was not a feasible proposition, since the Maple Machine provides no way for such a debugger to control another program. However, since the Maple Machine is just a program itself, it would be easy to write a debugger that was actually part of the Machine. This would effectively provide the facilities offered by an in-circuit emulator to hardware designers.

One time when a debugger is not of much use is when the hardware itself is suspected of containing bugs. After all, if the hardware cannot be trusted to run the program under test, it will, in all likelihood, not fair too well running the debugger either. Such situations usually result when a new computer system is being designed and implemented, and not when developing application level programs. In these cases, computer design engineers use a tool called an in-circuit emulator. This is basically a special stand-alone computer that is designed to replace the central processor of the system being tested. This computer emulates the central processor, usually at full speed, but allows the engineer to watch what is happening "from the inside".

The debugging feature that was added was a multiple level trace facility. The user could run the program, and indicate a range of Maple Machine addresses that should be traced, and the level of detail of information required.

When the lowest level of detail was selected, each instruction executed within the specified range would result in a display of the instruction's address, its mnemonic, and as much of the contents of the stack (starting from the top) as would fit on the remainder of an 80 column line. By tracing the instructions in memory immediately preceding the one that caused the segmentation fault, it was possible to see how the offending address value was computed in the first place. This in turn pointed out an error in the implementation of a Maple Machine instruction.

The next level of detail that could be selected was a trace of all procedure calls and returns. Whenever the flow of control was transferred to a procedure, a line of the form,

```
---> procname
```

would be displayed. When a RETURN or RETVAL instruction was executed, a line of the form,

```
procname <---
```

was displayed. In the first case, *procname* is the name of the procedure that was called, while in the second case, it indicates the name of the procedure to which control was returned. Since these procedures correspond directly to C functions in the Maple Kernel, it was easy to follow the flow of control. This facility could be enabled independently of the tracing range, so that the procedure call information would be displayed for all calls and returns, while the detailed instruction execution information would be displayed only for a selected range of instructions.

A final level of detail provided by the debugging facilities was a complete stack picture of all active procedures. In other words, whenever a procedure was called, the contents of the entire return stack were displayed in a useful form. This showed the addresses of the argument and local variable storage for each active procedure, and the number of procedures active at any given time.

An additional debugging facility was added to allow tracing of data accesses as opposed to instruction fetches. The user could specify a range of addresses that should be monitored for attempts to read from and write to them. This feature was implemented when it was discovered that some of the bugs resulted from data being written over the code image.

Several smaller run-time checks were built into the interpreter to trap certain errors and announce them, rather than letting the interpreter itself crash. Any attempts to address Maple Machine memory were checked for validity before being carried out. Overflow of the evaluation and return stacks was also monitored.

The various low level debugging facilities described in this section allowed the author to get Maple running on the Maple Machine within two days of the first successful compilation. There were of course still further bugs that surfaced when Maple was asked to perform more complicated operations, but most of the basic facilities worked. Performance was very poor, since at this point, no attempts had been made at optimization. The interpreter was also performing extensive checks with each instruction it was interpreting, resulting in a lot of overhead. When the debugging facilities were eventually removed, execution speed tripled.

6.4.2. Using The Assembler's Map and Correlation Files

In Chapter 5, it was mentioned that the assembler can produce two output files in addition to the code image file. These are the map file, and the correlation file. The first lists the name, type, and location of every identifier in assembly language input file, while the second indicates the correlation between assembly language source code lines, and Maple Machine instruction addresses in the code image file.

The map file is used by the debugging facilities to provide the procedure call information. Whenever a call occurs, the target address is looked up in the map file, and the corresponding name is displayed. When a RETURN or RETVAL instruction is executed, the return address is compared to each address in the map file until one is found that is larger than it. The previous identifier is then displayed as the name of the procedure to which control was returned.

The correlation file is not used by any of the debugging tools. It can be used by the user who, armed with the address of an offending instruction, wishes to find the corresponding line of the assembly language source code. Since the compiler inserts the C language source into the assembly language file as comments, the debugging information can quickly lead to the offending C source statement. Ideally, this information should be provided automatically by the debugger, but the effort required to implement such a facility was not deemed worthwhile.

System Optimization

Optimization of a software system is the processes of improving that system so that it takes less time and/or memory to run. The word "optimization" is a bit of a misnomer, as it implies that the system is improved to the point where no further improvements are possible. The word "improvement" would perhaps have been a better choice.

In a conventional system, optimization is usually performed by a language compiler on the code that it is generating. The term "optimizing" is often used by a compiler manufacturers to describe what that manufacturer's compiler does that no others do, so there exists a great range of optimizing compilers.

Compilers that really do perform some optimization other than just generating reasonable code in the first place, use techniques that fall into two general categories: global optimization and peephole optimization.

Globally optimizing compilers attempt to examine the structure of the program (or possibly an internal representation thereof) they are compiling, and detect such things as common sub-expressions which need only be computed once, or loop invariant code which can be moved outside of the loop since it is not affected by it. They also try to track the active life of variables, so that intelligent use can be made of the target machine's registers.

Peephole optimization is a process that is usually performed on either the compiler's intermediate representation of a program, the assembly language or binary output, or both. Peephole optimizers attempt to replace commonly occurring sequences of code with shorter or faster sequences that perform the same operations. The term "peephole" is used because the optimization is performed by examining small fragments of the code, with no regard to the overall structure of the program.

Many compilers perform both types of optimization. Global optimization is useful for overcoming deficiencies in the source language, such as the requirement to write something like,

```
A[I+J] = A[I+J] * 2
```

in FORTRAN, instead of the more efficient and expressive C construct,

```
a[i+j] *= 2
```

in which the subscript expression is evaluated only once. Global optimization can also reduce the effects of inefficient coding such as in the sample below, where the computation of *(row * column)* really only needs to be carried out once:

```
i = 1;
while(i < 1000)
      i = i + (row * column);
```

A good globally optimizing compiler can allow the programmer to concentrate on writing readable code, instead of on using obscure tricks for efficiency's sake. A peephole optimizer can further improve the code by making more efficient use of the target machine's instruction set without unduly complicating the compiler with details of special instruction formats. It also requires some intelligence on the part of a compiler to prevent the generation of such sequences as a *push* instruction followed immediately by a *pop* instruction; it is easier to simply generate such sequences, and have the peephole optimizer remove them.

In the Maple Machine system, there are even more opportunities for optimization. In addition to optimizing the compiler, one can optimize the implementation of the interpreter and the design of the instruction set. In addition, optimizations can be geared towards the specific program being compiled, namely the Maple kernel, since its execution is the sole reason for the existence of the Maple Machine.

7.1. Improving the Compiler

Most of the improvements to the Maple Machine C compiler were done before the first Maple Machine was ever implemented. The compiler generated some very inefficient code in some cases. Many of these inefficiencies vanished when the compiler was retargeted, since the stack machine architecture more closely matches the structure of the compiler itself. For example, the 8086 version of the compiler, when given the following

C source code,

```
i = 7 + 8;
i = j + 7;
```

generated the following 8086 assembly language instructions:

```
mov   ax,7
mov   bx,8
add   ax,bx
mov   _i,ax

mov   ax,_j
mov   bx,7
add   ax,bx
mov   _i,ax
```

A more sensible compiler would have generated a more efficient sequence, such as this:

```
mov   ax,15
mov   _i,ax

mov   ax,_j
add   ax,7
mov   i_,ax
```

The same C code when fed to a poor Maple Machine compiler would result in the following output:

```
7 8 +
i !

j 7 +
i !
```

Clearly, the second case cannot be optimized any further. The first case however would be more efficient if generated as:

```
15 i !
```

So, the first change made to the compiler was to implement compile-time evaluation of constant expressions. This was actually done before the first Maple Machine version of the compiler was produced, since many of the initializers in the Maple kernel source code made use of constant expressions (which are evaluated at compile time), and not just literal constants.

7.1.1. Maple Specific Functions

The Maple kernel performs several operations quite frequently. These operations are relatively simple, resulting in two to four instructions on a conventional machine, and so would suffer greatly from function call overhead if they were implemented as C functions. However, a function-like syntax was desired for these operations since the expressions performing them were rather obscure and meaningless when written out. Thus, they were implemented as macros, which are expanded by the Margay preprocessor during compilation.

Since these operations occur so frequently, at a cost of two to four instruction executions each, a savings in code size and execution time could be realized if each such operation could be replaced by a single instruction. This was done, and the resulting *?LENGTH*, *?ID*, and *?CASE_ID* instructions were discussed in detail in Chapter 4.

The definitions of these macros were removed from the macro definition files. The preprocessed code would thus be passed to the compiler with the macro invocations in their source form, appearing just like normal function calls. The compiler was modified to recognize the macro names as built-in operators, generate code to evaluate the argument, and then generate the single instruction that would perform that operation. These instructions were then added to the assembler and interpreter.

7.1.2. Improved Function Calling

Another area where the Maple Machine system was initially weak was the procedure (function) call mechanism. Every function call required the following steps:

1. Evaluate each argument, leaving the result on the stack.
2. Put the argument count on the stack.
3. Perform the function call, saving the program counter, stack pointer, and frame pointer.
4. Allocate the local variables using a *STACK* instruction.
5. Fetch the argument count (which was placed there by the calling code) from the stack into a local variable.
6. Execute the body of the procedure.
7. Deallocate the local variables using a *STACK* instruction.
8. Restore the program counter, stack pointer, and frame pointer, returning from the procedure.
9. Deallocate the arguments using another *STACK* instruction.

Besides requiring three *STACK* instructions to be executed for each procedure being called, there were several other side effects. C functions that took a variable number of arguments had to explicitly request the argument count from a pseudo-variable called "$#". This meant of course that the Maple kernel source would have to be extensively modified for those functions. All of these kernel functions used the contents of the first formal argument to determine the number of actual arguments that were passed, but the argument count was needed to determine the address of the first actual argument. This was because the Maple Machine C compiler evaluated function arguments from left to right, instead of the more traditional (but not guaranteed) right to left. As a result, the first argument was furthest from the top of stack, and its exact location depended on the number of arguments. Thus, the "$#" variable and the associated modifications to the code were required.

The author had considered modifying the compiler to evaluate function arguments from right to left, but decided that this would have been too difficult within the existing framework of the compiler. Instead, a new procedure calling mechanism was devised which would make the existing left to right evaluation compatible with any existing code, and also greatly speed up the function calling process. This modification consisted of adding a separate argument pointer register and changing the structure of procedure activation records.

The argument pointer register is analogous to the frame pointer register, except that it points to the first argument of the called function, whereas the frame pointer points to the first local variable. The compiler was modified so that argument references would be generated as positive offsets from the argument pointer, instead of the negative offsets from the frame pointer that were used before. New instructions were added to the Maple Machine to deal with this. These instructions (such as "#+") are described in Chapter 4. A fetch of the first local variable was thus always generated as,

```
0 #+ ?
```

(actually as *ARG_0_FETCH* when the shortcut instructions were later added), instead of,

```
-n $+ ?
```

where the value of *n* depends on the number of actual parameters. This completely eliminated the need for the "$#" variable, since the first actual parameter could always be accessed via the first formal parameter in the function definition.

The other change that was made was in the values stored in the procedure activation record. It was no longer necessary to store the current value of the evaluation stack pointer. On return from a procedure, the evaluation stack pointer is set to the value currently in the argument pointer register, effectively unstacking all the actual parameters and local variables of the procedure. This eliminated the need for two of the three *STACK* instructions previously required. The parts of the compiler that kept track of how many words to *STACK* and un*STACK* could thus be removed.

7.1.3. Other Optimizations

The changes to the compiler described above (except for compile-time evaluation of constant expressions) were primarily to correspondence to with architectural changes in the Maple Machine. The evaluation of constant expressions at compile time is a facility that any decent compiler should provide.

The Maple Machine C compiler does not perform any global optimizations. The reasons for this are twofold. First, Maple was originally written when compiler technology was not as advanced as it is today. The source code has basically been optimized by hand, often at the expense of readability. A fairly extensive optimizing compiler would be required to make much of an improvement. Secondly, the compactness of the Maple Machine instruction set makes it very easy for the compiler to generate small code images without having to resort to such complex measures. Even without the peephole optimizer described in the next section, the code image for the Maple kernel is only 30% larger than the code image generated by the 4.3 BSD UNIX C compiler.

7.2. An Assembler Peephole Optimizer

It is the author's contention that the Maple kernel would benefit little from a globally optimizing compiler. A peephole optimizer on the other hand is an absolute necessity, as the Maple Machine C compiler will often generate rather inefficient code for even the best written (from the compiler's point of view) C programs. This is due primarily to the fact that adjacent instructions in the output are often generated by widely separated parts of the compiler, which do not take context into account. As mentioned earlier, such inefficiencies are more easily dealt with by a peephole optimizer than by modifying the compiler.

7.2.1. Why Optimize in the Assembler?

Traditionally, a peephole optimizer is a part of a compiler. The compiler's intermediate code or object code output is examined by the optimizer while it is being emitted, and changes are made "on the fly". This is a suitable approach when the compiler is generating either machine instructions or some code with a fixed format. Performing such optimization on assembly language however would require a great deal of textual processing, which would degrade the performance of the compiler and make it unnecessarily complicated. For this reason, it was decided to make the peephole optimizer a part of the assembler instead.

When the assembler is generating a code image from an assembly language source file, it is simply depositing a sequence of bytes representing instructions into a large array. It is a fairly simple matter then to examine the last few bytes currently in the array for common sequences that can be replaced. The assembler merely calls the optimization routine after each instruction is emitted. The routine makes any suitable changes, updating the instruction pointer if the number of instructions was changed, and returns.

7.2.2. The Code Burst Table

Peephole optimization consists of examining an instruction sequence for specific "code bursts", short sequences of instructions performing some function, and replacing some of these with shorter bursts. The only difference between any two such optimizations is in the instructions involved. Thus, the process lends itself nicely to a table driven approach, in which a table simply stores each code burst that can be replaced, and the corresponding code burst to replace it with.

The Maple Machine assembler contains a table structure with two elements each. Each element is an array of bytes large enough to hold the longest code burst to be examined. The first array describes the code burst to be replaced, and the second describes the replacement. The code burst in each array is followed by a zero byte, to indicate where it ends. The last table entry is followed by a structure containing empty code bursts. Adding new peephole optimizations (or "peepholes", as they are commonly called) simply consists of adding a new row to the table and recompiling the assembler.

As each instruction is emitted, the last few instructions in the code image are compared with each code burst in the table. If a match is found, a replacement is made and the process repeated in case further optimization possibilities result. The table is currently searched by a linear search, but it is small enough to not warrant a more efficient search algorithm.

7.2.3. The Fast SWITCH Construct

In Chapter 4, we described the *SWITCH* instruction which is generated by the compiler for a C **switch** statement. This instruction is executed by traversing a list of *CASE* instructions, and comparing the associated constant with the value that was on the stack. When a match is found, the code following the *CASE* instruction is executed.

There is also an alternate form of the *SWITCH* instruction which does not have its own separate mnemonic. After assembling a *SWITCH* instruction sequence, the assembler determines the largest *CASE* value, the smallest *CASE* value, and the number of *CASE*s. If the range of *CASE* values is less than four times the number of *CASE* values, then the alternate form is generated. In the alternate form, a *GOTO* is generated after the *ENDSWITCH*, followed by the address of the first memory location after the jump table that is about to be created. The original *SWITCH* instruction is then changed to a special *FAST-SWITCH* instruction, and the word following it filled in with the address of this jump table. The jump table itself begins with two words giving the smallest and largest *CASE* values, and one word giving the address of the *DEFAULT* instruction. Following this is one word for each value in the range of *CASE* values giving the address of the *CASE* instruction corresponding to that value, or the address of the first instruction after the jump table if there was no corresponding *CASE* instruction. In order to keep the assembler as simple as possible, no attempt is made to go back and remove all the information that was filled in for a normal *SWITCH* statement. All the *CASE* to *CASE* pointers are still in place, although there is no longer a pointer from the *SWITCH* instruction to the first *CASE* instruction. The structure of an assembled *FAST-SWITCH* statement is shown in Figure 7.

When the Maple Machine encounters a *FAST-SWITCH* instruction, it pops a value from the stack and compares it with the smallest and largest *CASE* values. If it is out of range, it jumps to the *DEFAULT* instruction or the first instruction after the jump table. Otherwise it subtracts the smallest value from the popped value and uses the result as an index into the jump table to fetch the address of the corresponding *CASE* instruction.

7.3. Improving the Interpreter

The first implementation of the interpreter was written with one goal in mind: making it work. Time and space efficiency were not considered. Once everything was working however, it was desired to improve the performance. Although the VAX, on which the whole Maple Machine system was being developed, was not a potential target for portability (since Maple already runs on the VAX in its compiled form), it was deemed important to fine tune the interpreter's performance there so that some measurement of interpreted versus compiled speed could be made. This would allow some predictions for performance on potential target machines based on the speed of other compiled programs on those machines.

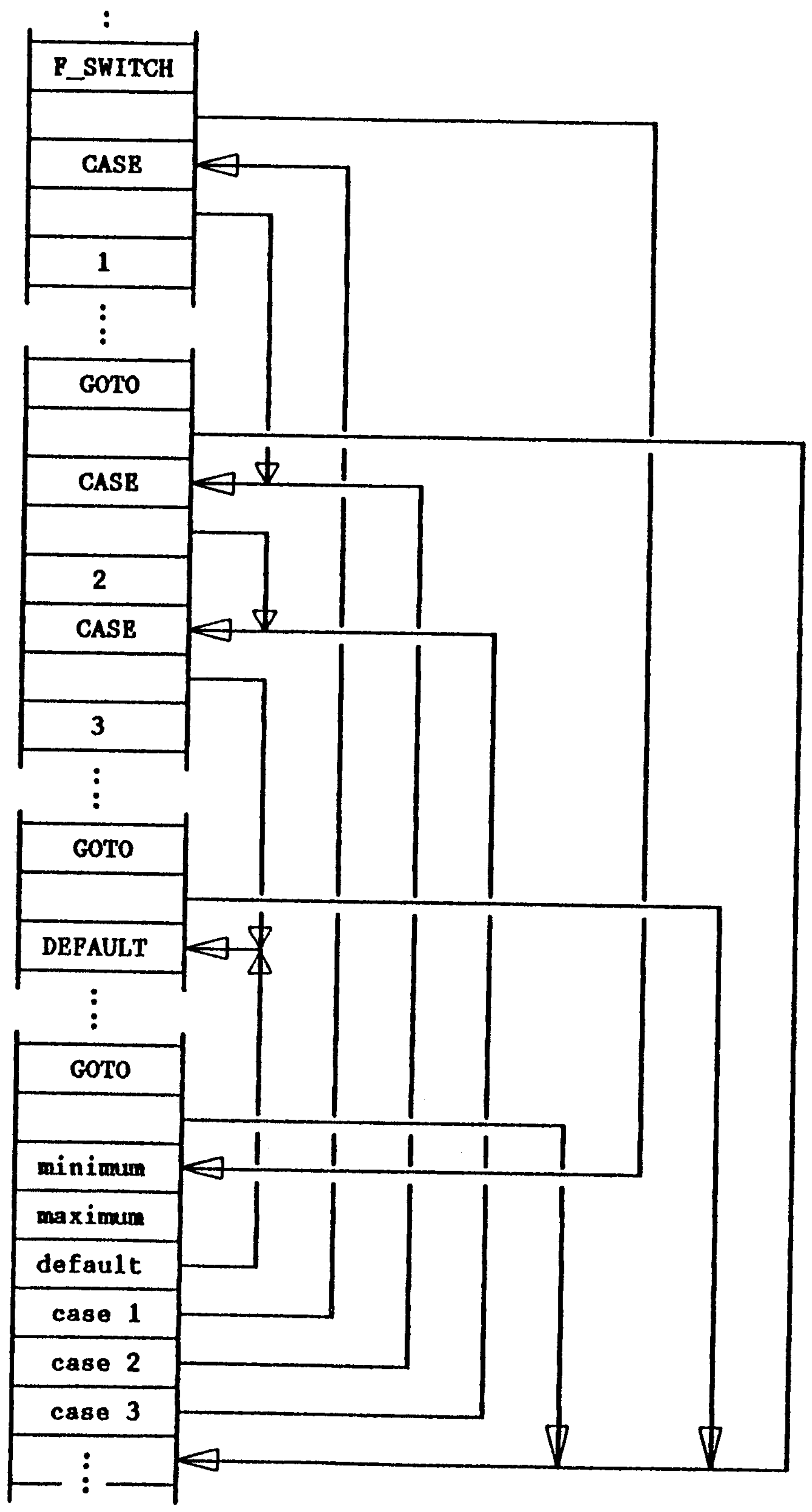

Figure 7. Structure of a FAST-SWITCH statement

7.3.1. Representation of Memory

In the first implementation, memory was represented as an array of bytes. The evaluation stack was represented as an array of integers occupying the same storage as part of the main memory. Accesses to the memory were written as C subscript expressions. For example, the following two lines of code fetch the next instruction, and access the value on the top of the stack respectively:

```
ir = memory[pc++];
temp = evalstack[evalsp];
```

Each subscript expression involves the addition of the index value to the base address of the corresponding array to compute the address of the desired value. Since the first line of code shown is executed for every instruction, and the code for most instructions includes at least two examples of the second line, there was a potential speed gain in using pointer variables, instead of arrays and subscripts. By changing *pc* to a pointer to a character, and *evalsp* to a pointer to an integer, the above two lines of code become:

```
ir = *pc++;
temp = *evalsp;
```

Of course, the code for those instructions that make references to memory relative to the beginning of the Maple Machine memory must add the offset of the memory array to any addresses. Once the addresses are converted to refer to real machine memory however, all subsequent references can be done without the additional overhead of a 32-bit addition operation.

7.3.2. Reducing Overhead

Every real machine instruction that is executed in the process of fetching a Maple Machine instruction results in an overhead time for each and every instruction executed. Even if this additional overhead is only one microsecond per instruction, it can quickly add up. For example, a particular collection of Maple test programs results in the execution of approximately 440 million instructions. If one microsecond was added to each instruction, this would result in an additional 440 seconds, or 7 minutes, of required CPU time. The time required to execute these tests with the current version of the interpreter is 1410 seconds, or 24 minutes, of CPU time. Thus, the additional microsecond per instruction would result in a 29 per cent performance loss.

To minimize this loss, the instruction fetching sequence was examined in great detail. The top of the interpreter's main loop originally looked like this:

```
for(;;) {
      ir = memory[pc++]
      switch(ir) {
```

The instruction being fetched was stored in the variable *ir* because one instruction needed to know the value of its opcode to extract a short constant from it. The first attempt at improving this resulted in:

```
for(;;) {
      switch(ir = memory[pc++]) {
```

Unfortunately, this did not result in any increased speed, since the VAX C compiler already generated the desired code from the original source.

Since the value of *ir* was needed only for the **switch** statement, and in the code implementing one instruction, it was decided that perhaps it was not worth saving. It could be recomputed by the one instruction that needed it, at a slight cost in speed for that instruction, but for a gain for all other instructions. Thus, the loop was changed to:

```
for(;;) {
      switch(memory[pc++]) {
```

The instruction, which originally referred to *ir*, was modified to refer to *memory[pc-1]*. The net result of this change was an increase in performance of about 10%, even though the affected instruction was one of the most frequently executed.

The final change made in this area was the change from an indexed array to a direct pointer into real machine memory as described in the previous section. This resulted in the following code:

```
for(;;) {
        switch(*pc++) {
```

Surprisingly, this resulted in only a one per cent performance increase, although the analogous change for evaluation stack access increased performance by about five per cent.

7.4. Adding New Instructions

Designing and implementing a virtual machine, such as the Maple Machine, gives the designer some unique opportunities generally not available to the designers of hardware machines. We have already mentioned the ability to debug by producing debugging information from within the machine. A far more important opportunity however is that of dynamically tailoring the instruction set to the code generated by the compiler, and even to the application program (the Maple kernel in this case) being compiled.

The problem was determining what instructions to add. Examination of the Maple source code pointed out some obvious candidates, such as the *?LENGTH*, *?ID*, and *?CASE_ID* instructions corresponding to the similarly named and often used macros. Further sequences that could be replaced by a single instruction could be discovered by examining the assembly language output of the compiler. Such an examination however will only indicate which instructions occur most frequently. What is of greater interest is the instructions which are executed most frequently. In other words, the dynamic instruction sequence is more important than the static instruction sequence. For example, the *STACK* instruction is only generated once for each function in the Maple source code, but is executed many times, since each function typically calls many others, often recursively. To analyse the dynamic behaviour of a program, it is necessary to keep a record of the instructions executed, and present this information in some usable form. This was done by means of execution profiling routines that were added to the interpreter.

7.4.1. Generating Instruction Profiles

Since each new instruction was to replace two or more existing instructions, it was not sufficient to simply keep track of how many times each instruction was executed. Instead, the number of times each **pair** of instructions was executed was required. One could this extend this to groups of three or more instructions, although this was not done due to time constraints; the generated reports would also be very large.

A modified version of the interpreter was created which would keep track of this information in a large array. With an initial instruction set size of about 60, this required an array of 3600 integers, or 14400 bytes. A larger array was used to allow expansion of the instruction set. As each instruction was executed, it and the previously executed instruction were used as indices into this array, and the corresponding element was incremented. When the interpreter completed execution, this array was written out to a text file, one row at a time.

Maple was run on the Maple Machine for several representative members of the Maple test suite, and execution profiles recorded for each one.

A reporting program was then written to make use of this information. All the profile files for one test run were accumulated, and a table was produced, containing one entry for each pair of instructions that was executed at least once. This table was then sorted by execution count so that the most frequently executed pairs could be found. The table is followed by a count of the total number of instructions executed in the test. Shown below is a sample of the last ten lines of such a table (the instruction mnemonics and percentages are not part of the output):

```
54, 0:     20318714     GOTO, push-tiny-constant      2.3%
58, 0:     22255884     POP, push-tiny-constant       2.5%
53, 0:     26900716     DUP, push-tiny-constant       3.0%
10, 1:     28504394     $+, !                         3.2%
 1,58:     29680517     !, POP                        3.3%
 3,42:     40380808     #+, ?                         4.5%
 0, 3:     40893398     push-tiny-constant, #+        4.6%
42, 0:     66518944     ?, push-tiny-constant         7.5%
10,42:     90453729     $+, ?                        10.2%
 0,10:    112539659     push-tiny-constant, $+       12.7%

889580206 instructions executed
```

The percentage figures indicate the percentage of number of instructions executed represented by each pair. The figures cannot simply be added up because each instruction is represented in the table many times. Adding up the percentages for the entire table would result in 200 per cent. Adding up the figures for the instructions listed above gives 53.8 per cent. Thus the instructions represented by the last ten pairs listed account for somewhere between 26.9 and 53.8 per cent of the total instructions executed (the figure of 26.9 is the worst-case, if each instruction in each pair in the list was also completely represented in the other pairs in the list). The excerpt shown above was from an early test run, before the Maple specific functions (*?LENGTH*, etc.) were added, and before any of the optimizations discussed below were done.

The profile reporting program also generates another output file giving a graphical representation of instruction pair occurrences. A sample of such an output (corresponding to the table above) is shown in Figure 8.

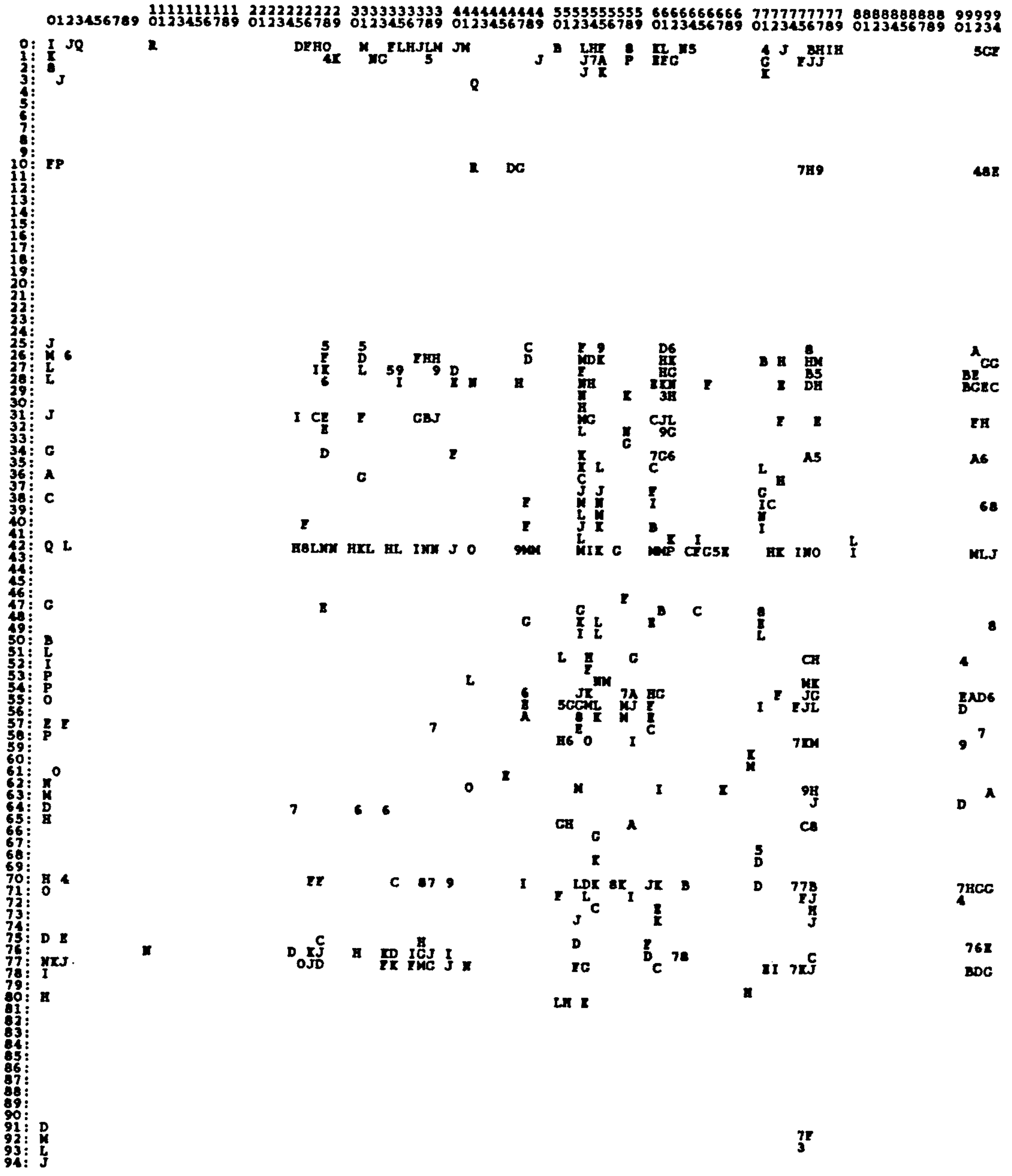

Figure 8. Instruction pair execution profile before optimization.

Each row represents the first instruction in a pair, while each column represents the second instruction in a pair. The letter or digit at the intersection of a row and column indicates the approximate number of times that pair was executed. The digit "0" corresponds to 1 execution , "1" to 2, "A" to 1024, and so on, doubling with each digit and then each letter.

Finally, the profile reporting program generates a table indicating how often each individual instruction was executed. For example, the last ten lines of the table corresponding to the instruction pair table shown earlier are:

```
55:      24981788     IF                        2.8%
62:      25062277     SCALE+                    2.8%
77:      26829691     push-long-constant        3.0%
 1:      38397976     !                         4.3%
53:      38553392     DUP                       4.3%
 3:      40893398     #+                        4.6%
58:      42387219     POP                       4.8%
10:     119171345     $+                       13.4%
42:     172675935     ?                        19.4%
 0:     187820516     push-tiny-constant       21.1%
```

Thus, these 10 instructions account for 81 per cent of all the instructions executed, even though they represent only 17 per cent of number of instructions (57) initially in the instruction set. This just confirms the widely held belief that about 20 per cent of the code in a program does 80 per cent of the work.

7.4.2. Analysis

Armed with the information provided by the profiling program, it is possible to determine which instruction pairs can be reasonably replaced with a new single instruction. There is even sufficient information to detect instructions that are never executed except as part of the same sequence.

Two sequences, one ending in a certain instruction, and the other beginning with that same instruction, might actually represent a commonly occurring sequence of three instructions. This is especially likely if the two sequences have approximately the same frequency of occurrence. It is even more likely if the frequency of occurrence of the instruction common to the two sequences is similar to the frequency of occurrence of those sequences.

For example, consider the following two sequences from the table presented previously:

```
0, 3:     40893398     push-tiny-constant, #+     4.6%
3,42:     40380808     #+, ?                      4.5%
```

The number of occurrences of the two is very close, and one might suspect that the sequence "push-tiny-constant, #+, ?" is quite common. On examining the individual frequency of occurrence of the "#+" instruction, we find the following:

```
3:        40893398     #+                         4.6%
```

Notice that this instruction occurs with exactly the same frequency as the sequence "push-tiny-constant, #+". This implies that this instruction occurs only as part of such a sequence. One can further conclude then that any "#+" instruction preceding a "?" instruction must also occur as part of a "push-tiny-constant, #+" sequence. Thus, of all occurrences of such sequences, 98.75 per cent are followed immediately by a "?" instruction. The few that aren't occur infrequently enough to not warrant further consideration.

Since this three instruction sequence starts with an instruction to load a tiny constant (one between 0 and 127 inclusive), it is not clear from the profile information which constants represent most of the occurrences of this sequence. However, it is known that this sequence fetches the Nth parameter of a function, and that most functions in the Maple kernel take three or fewer parameters. One might suspect that the tiny constants 0, 4, and 8 would thus represent most of these cases. Thus the instructions,

```
ARG_0_FETCH
ARG_1_FETCH
ARG_2_FETCH
```

were added to the instruction set to correspond to the sequences:

```
0 #+ ?
4 #+ ?
8 #+ ?
```

Examining the instruction pair profile table for the corresponding instructions to fetch local variables (as opposed to parameters), we see the following:

```
  0,10:    112539659        push-tiny-constant, $+       12.7%
 10,42:     90453729        $+, ?                        10.2%
```

The individual execution profile of the "$+" instruction is given by:

```
 10:       119171345        $+                           13.4%
```

This time, we can see that not all occurrences of "$+" occur after a "push-tiny-constant" instruction, although 94.4 per cent do. Similarly, 75.9 per cent of them occur before a "?" instruction. Thus, somewhere between 71.3 and 75.9 per cent occur as part of the three instruction sequence.

This sequence is used to fetch the values of local variables. Examining the Maple source code indicates that most functions have seven or fewer local variables, so the following instructions were added to the instruction set:

```
FRAME_0_FETCH        FRAME_3_FETCH        FRAME_6_FETCH
FRAME_1_FETCH        FRAME_4_FETCH        FRAME_7_FETCH
FRAME_2_FETCH
```

In addition to the *ARG_n_FETCH* and *FRAME_n_FETCH* instructions, corresponding instructions to store values in the local variables and parameters were added. Although these occur less frequently, they were deemed a source of possible performance increase.

7.4.3. Modifying the System

To add the new *ARG_n_FETCH*, *FRAME_n_FETCH*, *ARG_n_STORE*, and *FRAME_n_STORE* instructions to the instruction set, opcodes were assigned to them, and code was added to the interpreter for each instruction. Then, each instruction and the sequence that it was designed to replace was inserted into the peephole optimization table in the assembler. For example, the table entry to replace,

```
8 $+ ?
```

with,

```
    FRAME_2_FETCH
```

is:

```
    { {TINY(8),FRAME_ADD,FETCH,0}, {FRAME_2_FETCH,0} }
```

TINY is a macro which simply adds 128 to its argument, which gives the instruction opcode to load the corresponding tiny constant. All the other identifiers are constants representing the corresponding opcodes.

After modifying the assembler and the interpreter, both were recompiled. The Maple kernel assembly language file (produced by the compiler) was then reassembled by the new assembler.

7.4.4. Iterative Improvement

Once a few new instructions have been added to the Maple Machine, the profile tables become invalid. After all, many of the instruction sequences that accounted for most of the execution time have now been replaced by other instructions. Therefore, it is necessary to run all the profiling suite again to generate new profiles.

Theoretically, one could carry out this procedure indefinitely. In practice however, this is not practical. There will come a time when one of two things will happen. One will either run out of opcodes to assign to new instructions since there are only 128 possibilities, or further improvements will be negligible.

Initially, examination of the profile tables indicates that very few instructions account for most of the instructions executed. By reducing common sequences to individual new instructions, the frequency of execution of the sequence members goes down since many of their occurrences have been replaced by executions of new instructions. As a result, the distribution of execution occurrences of individual instructions becomes more uniform. This fact is visible pictorially in the graphic representation of the instruction profile. Compare the profile in Figure 9 below with the one in Figure 8.

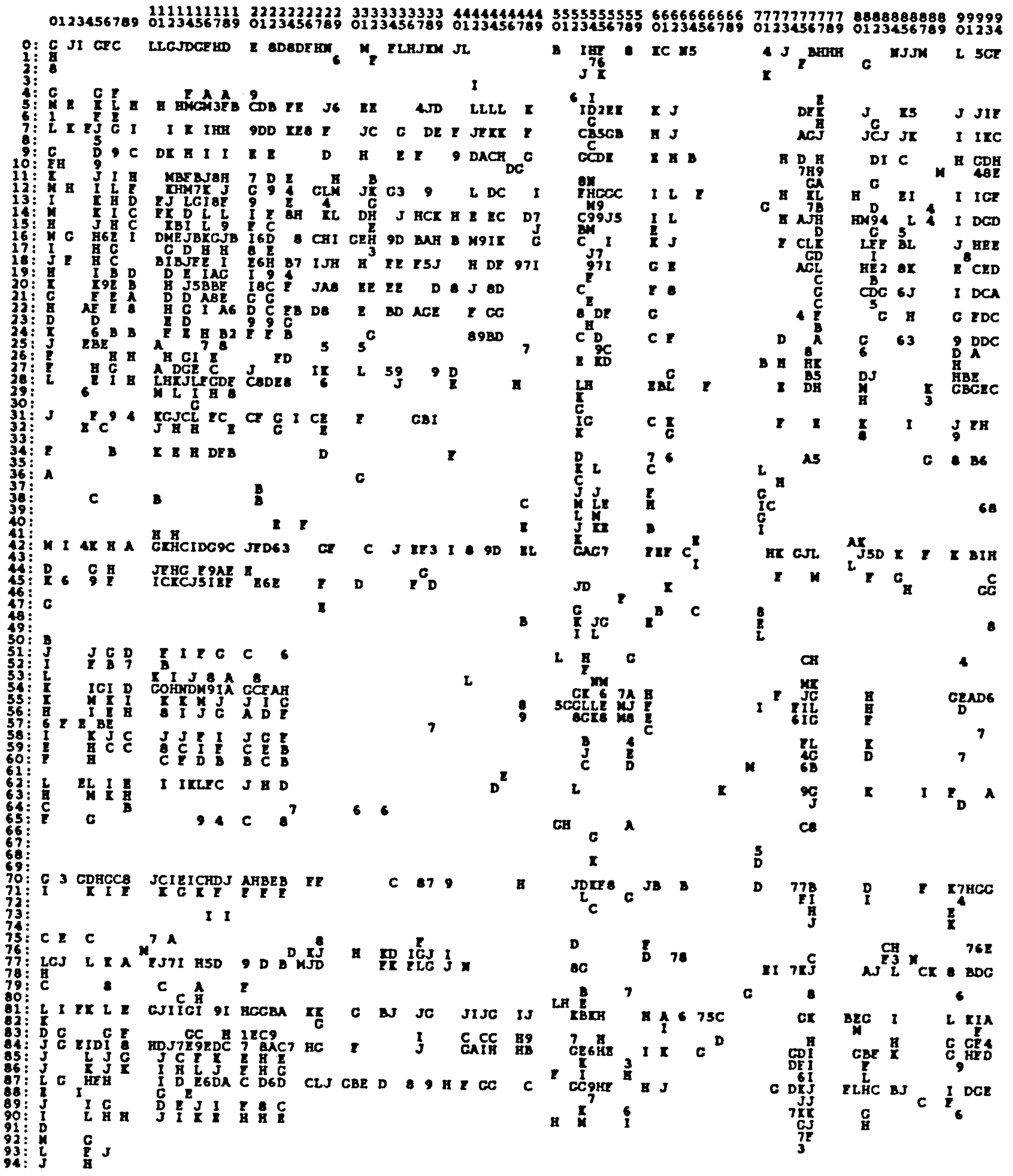

Figure 9. Instruction pair execution profile after optimization.

The first contains many distinct horizontal and vertical bands indicating commonly executed instructions, while the second is more uniform. The first was generated before the first new instructions were added, while the second was generated after four iterations of optimization. The wide white bands in the first represent the unimplemented instructions;

the profiles were normalized so that the same opcodes represent the same instructions in both profiles.

One can stop adding new instructions then when the effort in doing so begins to stop paying off. If no instruction or sequence of instructions accounts for a large percentage of execution time, then no single optimization will result in a large performance increase.

7.4.5. New Instructions

This section summarizes the new instructions that were added during the course of the iterative improvement discussed. These instructions are not categorized, since many of them perform operations that are a combination of several very different simpler instructions.

Each instruction is shown, followed by the sequence that it replaces. Many of these instructions replace many original instructions. These were often discovered when earlier new instructions appeared in common sequences with original instructions or other new instructions.

None of these instructions have a mnemonic, since they can only be generated by the assembler. They are never generated by the compiler, so the assembler does not accept them as input. Instead of a mnemonic then, the symbolic constant used within the assembler and interpreter source code is shown. The new instructions are described below.

```
FRAME_n_FETCH              4n $+ ?
ARG_n_FETCH                4n #+ ?
```

The *FRAME_n_FETCH* instruction is actually several instructions, for each value of *n* from 0 to 6. When one of these instructions is executed, the nth word in the local variable space of the currently active procedure is fetched and pushed on the stack.

Byte sized local variables use the long form, with a "B?" instead of "?" instruction. Only word sized variables that start on a word boundary, and fall within seven words of the beginning of the local variable space will be fetched using the short form. For example, in a C function with variable declarations,

```
int a; char b; int c; char d; int e; char f;
int g; char h; int i; char j; int k; char l;
```

the variables *a* and *i* will be fetched using *FRAME_0_FETCH* and *FRAME_5_FETCH* respectively. The remaining variables require the long form, either because they are byte sized, or they are word sized but do not fall on a word boundary. Since the large majority of procedures in the Maple kernel (and in most C programs) have less than 8 local variables, and these are almost always word sized, the longer sequence is rarely used.

The *ARG_n_FETCH* instruction, which is also several different instructions, is similar to the *FRAME_n_FETCH* instruction. Valid values for n are 0, 1, and 2. These instructions will fetch the nth argument of the currently active procedure.

All arguments are passed as words, so even byte sized actual parameters can be fetched by these instructions. The vast majority of procedures in the Maple kernel take less than four arguments, so the longer form is rarely used.

FRAME_n_STORE	4n $+ !
ARG_n_STORE	4n #+ !

The *FRAME_n_STORE* and *ARG_n_STORE* instructions are used to store values in local variables and parameters respectively. The same restrictions as discussed for *FRAME_n_FETCH* and *ARG_n_FETCH* apply.

ADD_FETCH	+ ?
SCALE_ADD_FET	SCALE+ ?
FRAME_ADD_FETCH	$+ ?
A_0_F_0_FETCH	0 #+ ? 0 $+ ?
F_0_SCALE_ADD_FET	0 $+ ? SCALE+ ?

These five instructions combine common sequences of fetching and address arithmetic. The *FRAME_ADD_FETCH* instruction in particular takes care of local variable fetches not covered by *FRAME_n_FETCH*. The *A_0_F_0_FETCH* instruction fetches the first argument and then the first local variable to the top of the stack. The *F_0_SCALE_ADD_FET* instruction is used for dereferencing a pointer to an integer with a subscript stored in the first local variable, a very common operation in the Maple source code.

`ROT_STORE`	`ROT !`
`STORE_POP`	`! POP`
`ROT_STORE_POP`	`ROT ! POP`

Each of these occurs quite frequently when a value is being stored indirectly through a pointer or array. The *POP* operation is often performed afterwards to discard the result. It should be noted that the code to implement *ROT_STORE_POP* is far simpler than the concatenation of the code required to implement the *ROT*, "!", and *POP* instructions, thus saving even more than just the overhead for two instruction fetches. Many of the other new instructions also exhibit such simplification, although to a lesser degree.

`IF_EQUAL`	`== IF`
`WHILE_GREATER_FOR`	`> WHILEFOR`

These represent the most common conditional test and loop termination conditions respectively.

`FUN_CALL_0`	`0 func-name`
`FUN_CALL_1`	`1 func-name`
`FUN_CALL_2`	`2 func-name`
`FUN_CALL_3`	`3 func-name`

These four instructions are used to call functions with zero, one, two, or three arguments respectively. They eliminate the need to push the argument count on the stack and pop it off again. These optimizations required a change to the body of the assembler, since the peephole optimizer was not powerful enough to deal with an arbitrary function identifier (address) as a recognizable token.

7.4.6. Other Optimizations

In addition to using the peephole optimizer to translate common code sequences into new instructions, it was also used to translate inefficient code sequences into more efficient ones. The compiler generally generates good code, but it does sometimes produce code bursts that are less than optimal, but are easily corrected by the peephole optimizer. Each of these additional optimizations is listed below, showing first the original code, and then the replacement code:

BNOT IF	IFZ
BNOT IFZ	IF
++ POP	POP
-- POP	POP
W+ POP	POP
W- POP	POP
VAL? POP	POP
1 +	++
4 +	W+

Opportunities for further optimization often result after one optimization has been applied. Whenever a peephole optimization is made, the entire table is tried again to produce more improvements. For example, after optimizing the sequence,

```
DUP   4   $+   !   POP
```

to the shorter sequence,

```
DUP   FRAME_1_STORE   POP
```

further optimization is possible. The following optimizations are those that can only result from earlier optimizations, since some of the instructions involved cannot be generated by the compiler:

DUP FRAME_0_STORE POP	FRAME_0_STORE
DUP FRAME_1_STORE POP	FRAME_1_STORE
DUP FRAME_2_STORE POP	FRAME_2_STORE
DUP FRAME_3_STORE POP	FRAME_3_STORE
DUP FRAME_4_STORE POP	FRAME_4_STORE
DUP FRAME_5_STORE POP	FRAME_5_STORE
DUP FRAME_6_STORE POP	FRAME_6_STORE
DUP ARG_0_STORE POP	ARG_0_STORE
DUP ARG_1_STORE POP	ARG_1_STORE
DUP ARG_2_STORE POP	ARG_2_STORE

7.5. Some Performance Figures

To evaluate the performance of the Maple Machine at various stages during its development, a subset of the Maple test suite was run. This is the same subset that was used to perform the profiling. During performance evaluation, the interpreter was first recompiled with profiling removed.

The results of the performance tests, along with the number of instructions executed during the corresponding profile run, and the size of the code image, are summarized below at several points during the Maple Machine's development. These points were:

1. First non-debugging run, with no peephole optimization.
2. After adding the special instructions *?LENGTH*, *?ID*, and *?CASE_ID*.
3. After adding the argument and local variable access shortcut instructions such as *ARG_2_FETCH*.
4. After adding all other new instructions and other peephole optimizations.

In addition, the times and code image size for compiled Maple running the same tests on the same machine (VAX 8650) are also given. CPU times are rounded to the nearest 100 (except for the compiled time), due to inaccuracies caused by varying system loads.

TEST RUN	TOTAL CPU TIME	CODE IMAGE SIZE	INSTRUCTIONS EXECUTED	SPEED RATIO re COMPILED
1	3200	207,100	889,580,206	65:1
2	3100	198,548	862,939,589	63:1
3	1800	173,920	572,889,565	37:1
4	1400	156,380	441,301,887	29:1
COMP	49	163,840	N/A	1:1

It appears from the chart that there is a very close relation between code size and number of instructions executed. A linear regression test on the results, with code size as the independent variable, and instructions executed as the dependent variable reveals a correlation coefficient of 0.99, indicating an extremely close correlation.

A Real World Example: The Oberon System

In December of 1988, the author was asked to port Maple to the Ceres workstation running the Oberon operating system. Ceres and Oberon were developed by a group headed by Professor Niklaus Wirth at the Eidgenössische Technische Hochschule (ETH) in Zürich, Switzerland.

8.1. A Brief Introduction to Ceres and Oberon

Ceres is a powerful single user work station based on the National Semiconductor 32032 CPU. It features a high resolution bit-mapped display screen, two megabytes of memory, a 40 megabyte fixed disk, a floppy diskette drive for backups, a network interface, and a keyboard and mouse. For further details, the reader is referred to the report entitled "Hardware Description of the Workstation Ceres" [6].

Oberon is the operating system for the Ceres workstation. Its development was begun in late 1985 by Professors Niklaus Wirth and Jürg Gutknecht. The goal was "to develop an operating environment especially tuned to personal workstations, and a language to implement the system" [25].

Oberon presents itself to the user as a multitasking multi-window environment, although in fact there is only ever a single process running. The user issues commands, which are directed at the process associated with a selected window (a *viewer* in Oberon terminology). The contents of a viewer are persistent (they do not scroll away, and the viewer does not vanish unless the user closes it), and can be edited and re-used as input. There is no distinction between input and output in the traditional sense, where input is what the user types, and output is what the program displays. Both input and output are all part of an editable text under Oberon. Procedures are invoked, and operate on the text that was last selected with the mouse, or on the contents of the viewer that was last *focused* on. Since viewer contents are persistent, and one can have multiple viewers on the screen at one time, the user can be pursuing several tasks at one time, directing commands at the different processes. When a process completes a command, the user can then issue a different command to that process, or to another. In the words of the system's designers, Oberon can be considered a "single process multitasking system." The user plays the role of task scheduler that is typically performed by the kernel of other multitasking operating systems.

One of the design goals of Oberon was to make it user-friendly. According to the designers, one of the greatest impediments to this goal is that of hidden states. A hidden state is one where the user is expected to perform some action that is different than the normal actions allowed by the system. The state is termed "hidden" because there is no real indication that the system is in such a state. A good example familiar to many readers is the system of modes used by many text editors, such as the UNIX *vi* editor. In one mode, the keys on the keyboard are used to move the text cursor around, while in the other mode, the keys are inserted into the text. It is impossible to tell by looking at the screen what mode or state the editor is in. This problem is brought about by the fact that control of the keyboard is shared among two or more tasks (not necessarily concurrent ones). When a text editor is in cursor movement mode, the cursor movement task is handling keyboard input; when it is in insertion mode, the text entry task is handling the input.

To prevent such modes or states, Oberon applications never seize control of the keyboard or mouse. All keyboard input results in the typed characters being inserted into the text in the viewer that is currently active (contains the text cursor). An application task is only given control when specifically invoked, perhaps with some marked region of the text as an argument. A programmer's text editor for example is simply a collection of commands that can be invoked to operate on the text in a viewer. It is not a program that actually controls such a text, or receives input from the user. Thus, the user can perform any other tasks between invocations of text editing commands. There is no need to write a complex text editor capable of dealing with multiple files, because the user can simply invoke the text editor commands on a different text in a different viewer. One can even have multiple views of the same text by opening another viewer on that text.

There is only one officially sanctioned exception to this rule. When the user wishes to log onto the network to obtain access to network services, such as the laser printer, a password must be typed after invoking the Net.Server command. The process that reads the password does take over the keyboard, since it would not be secure to type the password on the screen, highlight it, and then invoke the command. In an ideal world where security was unnecessary, even this exception could be removed, as passwords would not be needed.

Oberon consists of a collection of modules, each of which contains data types, constants, variables, and procedures to operate on some class of objects. These modules are written in the language Oberon [26], which is a successor to Modula-2 [24]. Several modules comprise the "inner core" of Oberon, which provides resource management, fixed disk drive management, file directory handling, file I/O, and module loading facilities. To these are added an "outer core" which provides modules to deal with texts, pictures, and viewers. A program written for Oberon can import any of these modules, and can then access the public parts thereof.

8.2. Obstacles for the Oberon Port

There were several restrictions imposed on the porting of Maple to Oberon.

First and foremost, there is no C compiler available for the Ceres, although the hardware architecture is capable of supporting a C implementation that could run Maple. The designers of Oberon consider reliability very important, and consider the lack of type safety and run-time checks in C to make reliability impossible to ensure. Thus, the port had to be done by way of the Maple Machine.

The second restriction was that the author of this thesis had to perform the port in Switzerland in the space of about two weeks. If it was not completed in that time, he could not simply go back the following week and continue!

Fortunately, the Maple Machine approach proved up to the task at hand, and the port was completed successfully. Some of the obstacles encountered along the way are outlined in the following sections:

8.2.1. The User Interface

As was described earlier, the user interface methodology of Oberon is drastically different from that of UNIX and similar systems. This seemed at first to be the largest obstacle, since Maple was written with the more traditional user interface in mind.

The Oberon version of Maple would have to be able to accept input that was sent to it as an argument (by selection) when invoked, and retain its state between such invocations (otherwise it would essentially be a batch processing version of Maple).

8.2.2. The File System

The Oberon file system is also very different from the UNIX file system. There is no concept of a directory hierarchy. All files are contained within one large directory, just like the names in a telephone book. Maple's libraries on the other hand are stored in a complex hierarchy of directories, with the procedures for various sub-functions stored in directories below those of higher level functions.

Some means would have to be found to map the Maple library path names to the file names of Oberon. The Maple kernel already contains an internal function to perform such mapping on machines that do not have a directory structure. This function must be rewritten whenever Maple is ported to such a machine by traditional means. In this case however, the responsibility for translating the names would have to lie with the Maple Machine, since the goal is to be able to run the same executable on all host machines.

There are also differences between the file I/O functions provided by UNIX, and those provided by Oberon. Under the UNIX model of file I/O for example, the process of creating and writing a new file is briefly summarized by:

1. Create the file and give it a name.
2. Write the data to the file.
3. Close the file.

Under Oberon on the other hand, the process is slightly different:

1. Open a nameless file for writing.
2. Write the data to the file.
3. Register the file by giving it a name and closing it.

In addition, Oberon distinguishes between texts and files. A text is simply a stream of characters which may or may not be visible in a viewer. A file on the other hand is a stream of bytes. When a text is stored in a file, various header information regarding fonts is stored at the beginning of the file. Thus, a Maple procedure reading a file containing text must skip this header information, while one reading a binary file must read the entire file.

8.2.3. The Oberon Programming Language

The last obstacle was again the lack of a C compiler. The only existing Maple Machine implementation was written in C for a VAX running 4.3 BSD UNIX. If Oberon had a C compiler, the Maple Machine would be fairly easy to port. Of course, Maple itself would be easy to port directly then too.

The Maple Machine would have to be ported by rewriting it in the Oberon language. The Machine was intended to be compact and easy to rewrite on any conceivable target machine, so this would be the real test of this contention.

8.3. Porting the Maple Machine

The first step in porting the Maple Machine was to become intimately familiar with the details of the Oberon operating system. It was radically different from anything the author had used before, and so seemed a formidable task. Much to the author's surprise, he had a very intuitive understanding of the system after only one day! The credit for this lies with the designers of the system however. It is the clear, simple design of the system, coupled with the excellent documentation that they have produced, that made this feat possible. Others (particularly in industry) could learn much from their example. By the end of the first day, a rough sketch of the code required to completely hide the Oberon interface from the Maple kernel running on the Maple Machine had been made.

8.3.1. The CFileIO Module

The first thing that was actually implemented was a module called CFileIO. This module exported several functions with semantics similar to the file I/O functions provided by the UNIX C run-time library. All the differences in file systems were hidden therein.

File naming convention differences were resolved by a simple algorithm. Each Maple library file name, possibly containing several directory names, was passed through a filter which removed the "/" characters separating the directory names, and capitalized the first letter after each such character removed. The resulting name was unique among all the other Maple library file names, and less than 32 characters in length, which is the limit imposed by Oberon. Any call to the functions in CFileIO which dealt with a file name could be passed a UNIX style name. This name was converted before the file was actually opened for reading or writing. A notable side effect of this algorithm was that Oberon style file names would pass through intact, allowing Maple users to use native file names.

The difference between text and binary files was also easily resolved, although only in the context of Maple. All binary files that Maple opens end with the two characters ".m". The CFileIO routines responsible for opening files thus examined the file name to determine its type. If the file was a text file, any header information in the file was parsed and skipped. Text files opened for writing were treated the same way as binary files, since text files did not **require** this header information.

Once the CFileIO module had been completed, a small test module was written (in Oberon) to use these procedures.

8.3.2. Writing the Interpreter

The next step was to implement the Maple Machine proper. Before leaving for Switzerland, the author had written a version of the Maple Machine in Modula-2. No suitable Modula-2 compiler was available for testing, but the program was syntactically correct as verified by a Modula-2 compiler on an IBM Personal Computer.

A first attempt was made to implement the Maple Machine by simply translating the Modula-2 program, since Oberon is generally a subset of Modula-2 (but with many improvements) [27]. It was quickly pointed out however that such an implementation would be quite slow. This was because the Modula-2 version used arrays to represent memory, and all array accesses under Oberon involve a run-time range check. Further inquiry with Professors Wirth and Gutknecht revealed that there were several standard (but undocumented) procedures recognized by the Oberon compiler. Each of these would actually generate a single machine instruction, such as a memory load or store. With this new information available, it was decided that a far better approach would be to translate the C version of the Maple Machine, which already worked in terms of pointers instead of arrays. Despite the vast differences in syntax, this actually turned out to be quite simple to do as a result of these special standard procedures.

8.3.3. Intrinsic Functions

Once the entire basic instruction set had been implemented, the next stage was to implement the intrinsic functions. Some of these were quite trivial to implement, since a corresponding function existed in the Oberon library. For example, *abs()* was one of these, which corresponds to the Oberon standard procedure ABS.

The file I/O functions were also straightforward to implement, since they could be written in terms of the already completed CFileIO module. The various string handling functions used by the Maple kernel, such as *strcat()* are very well defined, and were easily implemented in Oberon.

The more difficult functions to implement were those concerned with system specific features, and those concerned with the user interface.

The system specific functions supported by the Maple Machine are *exit()*, *getenv()*, *isatty()*, *time()*, *times()*, *signal()*, and *system()*. The functions *sbrk()*, *setjmp()*, and *longjmp()* are system specific when performing conventional ports, but their definitions have been completely fixed within the context of the Maple Machine.

The *exit()* function had to be implemented quite differently from the implementation under UNIX. No Oberon task ever really exits. Whenever a task completes a computation, it returns control to the user, but its state remains fixed. It is as if the task were suspended. This ties in closely with the user interface discussion later in this section. The *exit()* function of the Oberon Maple Machine simply sets a flag that indicates that Maple should be restarted the next time it is invoked, rather than resumed.

Oberon does not support the concept of environment variables. Thus, the *getenv()* function was implemented to return a pointer to a Maple Machine address containing a zero byte, which indicates that the desired environment variable was not found. Maple deals gracefully with this situation, since environment variables are used to specify optional parameters.

The function *isatty()* is used under UNIX to determine that the specified I/O channel has not been redirected. In other words, data written to or read from the associated file will be displayed or received from the user's terminal. There is no concept of redirection under Oberon, since all output goes to a text, which can be stored in a file later at the user's discretion. Thus, the function *isatty()* was implemented to always return true for the standard input, standard output, and standard error file descriptors. Refer to the discussion on user interface issues below for more details.

Oberon does not provide a function to return the number of seconds of CPU time, or the number of seconds since some arbitrary date. Thus, a new module, called TimeInfo had to be written to support the implementation of *time()* and *times()*. The only public procedure in this module simply reports the number of seconds since 00:00:00 on March 1, 1980. An internal procedure hides the details of accessing the Ceres' real-time clock chip. The *time()* function merely returns the current return value of TimeInfo.GetTime. The *times()* function also computes this value, and subtracts from it the time that current Maple process was started. The result is then multiplied by 60 to give the number of 60ths of a second of CPU time used. Whenever the Maple process is suspended, the total CPU time used so far is saved, so that the time between invocations does not contribute to CPU time. Thus, *times()* will always return the number of seconds that the CPU has actually spent running the Maple Machine since the last time the Maple was restarted.

Under UNIX, when the user presses CTRL-C, a signal is sent to the foreground process. If that process has not set up a signal handler, it is aborted, and control returns to the shell. If on the other hand it has set up a handler, that handler is given control. Maple under UNIX sets up a handler to catch the CTRL-C signal, and abort the current computation, returning the user to the Maple prompt. Oberon does not provide a signalling system, so a different approach was taken. Instead, the Maple Machine calls the keyboard handler whenever a function call is made, to determine if any keys have been pressed. If so, it seizes control of the keyboard just long enough to read that one key, and transfers control to the signal handler whose Maple Machine address was specified by a call to the *signal()* function. The notion of no hidden states is not violated, since the Maple Machine only seizes control of the keyboard to read one key, and then only after the key has been pressed; the machine never waits in a hidden state.

The last system specific function, *system()*, was simply implemented as a null operation. It displays the message,

```
maple: function system() is not implemented
```

in a system status viewer. This function is only ever called if the user asks Maple to execute an operating system command via the "!" directive. This is not necessary under Oberon, since the user can execute any operating system command at any time.

8.3.4. Hiding the User Interface

The last remaining task was to hide the user interface methodology of Oberon from programs running on the Maple Machine.

Output was straightforward to implement. The intrinsic functions that might be called upon to write to the standard output, such as *printf()* and *fprintf()*, were implemented to write to a text in a viewer. The appropriate calls to create such a text and viewer were inserted into the main procedure in the Maple module. Every time this procedure was invoked, it would first check to see if the viewer exists (it may not have been created yet if this was the first invocation, or the user may have closed it), and then create the appropriate viewer if not.

Input was the more complicated portion to implement. In a typical Maple implementation, the Maple kernel makes a call to the run-time library function *getc()* to read a character from the standard input. The library function will read such a character, and return it to the caller. Procedures under Oberon on the other hand never wait for input. Instead, they are passed input as an argument or as the most recently highlighted block of text when they are invoked. They then process this information and stop executing, returning control to the user.

To reconcile the Oberon approach with the more conventional approach used by Maple, a special procedure called *ExecuteGetC* was written. This function would retrieve a character from the last argument passed to Maple, and return it in the return value register of the Maple Machine. If on the other hand there were no more characters in the argument, *ExecuteGetC* would return FALSE. The Maple Machine would then set an internal flag indicating that further input was pending, and then terminate execution. When Maple is next invoked, it will notice that the flag is set, call *ExecuteGetC* again to fetch the first character in the argument to the new invocation, and then continue executing where it left off. This is all made possible by the persistence of state between invocations of commands. A simplified version of *ExecuteGetC* is shown below:

```
PROCEDURE ExecuteGetC(): BOOLEAN;
  VAR
    c: CHAR;
  BEGIN
    IF (stdinText = NIL) OR (Texts.Pos(reader) >= stdinTo)
    THEN
      RETURN FALSE
    ELSE
      Texts.Read(reader,c);
      retVal := ORD(c);
      RETURN TRUE
    END
  END ExecuteGetC;
```

The identifier *stdinText* refers to the text in which input was last highlighted using the mouse, while *stdinTo* is the index into that text of the first character after the highlighted text. The actual implementation of *ExecuteGetC* is more complex, since it attempts to supply the trailing semicolon and NEWLINE characters required of Maple input. This is so that an expression can be selected from the middle of a sentence in a document, for example, and passed as input without having to edit these extra characters in first.

In the CASE statement responsible for executing the intrinsic functions, the code for the *getc()* function is as follows:

```
PROCEDURE ExecuteIntrinsic(): BOOLEAN;
   BEGIN
       :
       :
      CASE ir OF
       :
       :
      | GETC:
         IF argList[0] = STDIN THEN
            IF ~ExecuteGetC THEN
               inputPending := TRUE;
               RETURN FALSE
            END
         ELSE
            retVal := CFileIO.FGetC(argList[0])
         END;
       :
       :
      END;
      RETURN TRUE
   END ExecuteIntrinsic;
```

What this does is first check to see if the input is requested from the standard input. If so, *ExecuteGetC* is called to put a character in *retVal*. If this fails, due to lack of input, the *inputPending* flag is set, and the procedure ExecuteIntrinsic returns FALSE.

Finally, the procedure *Maple.Evaluate*, which is invoked by the user wishing to evaluate an expression, looks like this:

```
PROCEDURE Evaluate;
   BEGIN
    :
    :
   (* if Maple has not been reset, then restart it *)
   IF ~mapleReset THEN
      LoadBinary;
      sigHandler := 0;
      retStackPtr := retStackBasePtr;
      argPtr := evalStackBasePtr - 2 * WORDSIZE;
      framePtr := evalStackBasePtr;
      evalStackPtr := evalStackBasePtr;
      inputPending := FALSE;
      startTime := 0;
      CFileIO.CloseAll;
      mapleReset := TRUE
   END;
   startTime := TimeInfo.GetTime() - startTime;

   (* find the most recently highlighted selection *)
   FindSelection(stdinText,stdinFrom,stdinTo);
   IF stdinText # NIL THEN
      Texts.OpenReader(reader,stdinText,stdinFrom)
   END;
   IF inputPending THEN
      IF ExecuteGetC() THEN
      END;
      inputPending := FALSE
   END;

   (* main interpreter loop *)
   LOOP
```

```
            SYSTEM.GET(pc,ir);
            INC(pc);
            CASE ir OF
             :
             :
            | INTRINSIC:
               IF ~ExecuteIntrinsic() THEN EXIT END;
             :
             :
            END
         END;
         startTime := TimeInfo.GetTime() - startTime;
      END Evaluate;
```

Using this approach, it was possible to run Maple on the Oberon Maple Machine without so much as recompiling it. The same executable that ran on the Maple Machine on the VAX also runs under Oberon. All of Oberon's user interface philosophies were adhered to, without changing the Maple source code at all!

Once Maple was running successfully, it was recompiled however to insert a serial number, and to change the user prompt. The default user prompt is simply the ">" character. What the user types then appears after the prompt. Since the user does not type at the prompt when Maple runs under Oberon, this prompt simply appeared at the beginning of each line of Maple output. This resulted in the first line of output of each calculation being shifted right by one character position. The prompt was changed to be a ">" character followed by a linefeed, so that output would begin on the next line instead.

8.3.5. Running Maple Under Oberon

Once the entire Maple Machine had been implemented, initial tests began. The first task was to transfer the Maple executable, as compiled by the Maple Machine development tools, to the Ceres. Due to the developmental nature of the Ceres system, tools for moving large quantities of data between a Ceres workstation or network and the outside world were not well refined. The transfer of the Maple executable was eventually accomplished by converting it into a text file using a UNIX utility called *uuencode*, splitting the resulting file into about eight pieces, and then sending them to the Ceres workstation by electronic mail. A graduate student at ETH had written a *uudecode* utility for Oberon, so that it was then a simple matter of recombining the separate parts and converting them back into a binary file.

The first attempt to run Maple on the Oberon Maple Machine resulted in a system crash. This was not surprising, since the Maple Machine was a fairly complex thing to write as one's first program on a new machine under a new operating system and in a new language. The crash was eventually traced to an error in the garbage collector, which would not return if one requested more than half the available storage. After reducing the size of the request, another attempt was made. This time, the Maple logo actually appeared before the system crashed. This was a better success rate than the first attempt on the VAX, due in part to the fact that the compiler and assembler were already debugged at this point, and the executable was known to be valid.

Oberon currently does not provide any extensive debugging facilities. All that was available was a fairly detailed symbolic dump of all global and local variables in a given module. Fortunately, this information immediately pointed out some errors in the Maple Machine implementation. A number of errors were fixed, and a careful code reading uncovered some more errors, which were also fixed.

The next attempt at running Maple actually succeeded, for a while. The logo appeared, the prompt appeared, and Maple could be asked to perform calculations with real numbers and polynomials. This milestone occurred only one week and one day after the author's arrival in Switzerland, and served to demonstrate the viability of the Maple Machine approach to portability; most conventional ports of Maple take longer than that.

8.3.6. Porting the Library

Maple uses a large library of routines written in the Maple programming language to perform most of its calculations. The only capabilities built into the kernel are arbitrary precision arithmetic, simple polynomials, differentiation of simple expressions, and the ability to interpret the Maple language. So, in order to be very useful, the library has to be ported as well.

Normally, porting the library is a straightforward task. It is simply a matter of copying the files to the new machine, and installing them in the appropriate directory. This procedure presents two obstacles to the Oberon port however: there were no tools for copying large files to the machine, and there are no directories.

The latter problem, that of directories, had already been solved by the CFileIO module. The path names of each file were simply converted to one long unique file name. It is the former problem that presented the biggest obstacle.

The Maple kernel was relatively easy to copy to the Ceres machine, since it was only about 160 kilobytes. The library files on the other hand comprise about 5 megabytes of source code. This is about one and a half orders of magnitude larger, and therefore that much more prone to error during any transfer process.

The first thing that was tried was to electronically mail the files over. Each file is relatively small, and the *uuencod*ed version would only be about 33 per cent larger. Unfortunately, this quickly failed, as the Ceres network mail system was brought to its knees by the deluge of messages. A second attempt was made by writing an archiving program to concatenate the files into a text archive, and then mailing large pieces of the archive. This also failed due to a maximum message size of about 50 kilobytes imposed by the mail system; over 100 messages would still have been required.

It was clear that the mail system would be inadequate for the job of transferring that much information in so little time. After spending a day trying, the author decided on a different approach. The archiving program was used again to put all the library files in one large archive. Then, the author wrote a terminal emulation and file transfer utility for the Ceres, and connected it to a standard terminal port on the network connecting all the terminals and computers at ETH. This allowed the archive file to be transferred directly from the VAX to the Ceres. Even this failed on the first attempt due to a limitation of 2.6 megabytes on Oberon files. A second attempt was made using smaller archives, and all the files were finally transferred.

A dearchiving tool was written in Oberon to extract each file from the archive, change the name according to the algorithm embodied in CFileIO, and write the file to the disk. This flurry of file activity resulted in a minor disk crash, which would have been a major set back, except for the heroic efforts of a graduate student who managed to reconstruct the file system.

Finally, the port was complete! By this point, almost another whole week had passed, and little time was left for extensive testing. A random selection of tests from the Maple test suite was transferred to the Ceres, and all tests except one ran flawlessly. The one that failed ran out of memory since the bug in the Oberon storage allocator would not allow enough memory to be allocated. This bug was being fixed at the time, and the amount of memory being requested by the Maple Machine can probably be increased.

8.4. Performance

During the course of implementation, the author had some doubts about the potential performance of the result. However, the inner loop of the interpreter was written almost entirely in terms of the standard procedures SYSTEM.GET and SYSTEM.PUT, which each generate a single instruction. Examination of the code produced by the compiler revealed that one could not have done much better in assembly language. In fact, Professor Wirth calls this style of programming, "assembly language programming with Oberon syntax", which indeed it was.

There was not enough time to run all the tests, so no direct timing results were available for comparison with the tests used in evaluating the performance of the VAX version. However, the handful of tests that were run under Oberon were later run on the VAX version as well. The general results indicated that the Oberon version required about twice the CPU time as the VAX version. Since Oberon is a single tasking system though, the Maple Machine gets 100 per cent of the CPU time when it is running, whereas a much smaller fraction of CPU time is available to a program running on a VAX. With a medium load on the VAX (a load average of about 3, as reported by the *loadaverage* command), the Oberon version of Maple required only about half the real time as the VAX version to run the same tests.

The subjective opinions of observers at ETH were also favourable. Many were impressed by Maple's speed. None had ever witnessed Maple in its compiled form.

All in all, the port of Maple to the Ceres workstation was a success. The vast differences in operating system architecture and programming methodology were easily overcome, and it was possible to run the same Maple executable code image on both the

Ceres and the VAX. Work is currently underway at ETH to port the Maple Machine to the new release of Oberon. A faster Ceres workstation, the Ceres-2, with about seven times the CPU speed, and four times the memory, is also being introduced, which should result in very good performance for Maple.

Future Directions

Two years of work developing the Maple Machine concept, and writing the development tools for it, has proven that it is viable. The machine is operating at a high enough level that the loss of performance due to interpretation is not too severe. At the same time, the machine is also simple enough to be quickly and easily ported to new hardware and operating system environments.

The design of the Maple Machine has not been frozen. New techniques for performance improvements are being examined. These include the addition of more complex instructions, different interpretation techniques, optimization at a higher level, and incorporating some small but important parts of the Maple kernel into the Maple Machine.

9.1. The IBM Personal Computer Family

The next task facing the author is to complete the port of Maple to the IBM Personal Computer family. Since the project was started, this family has expanded to include new machines with new capabilities. The three main classes of IBM Personal Computers are those based on the 8088 or 8086 CPU, those based on the 80286 CPU, and the 80386 based IBM Personal System/2 series. Each one is upward compatible with its predecessors, but each provides features that would be beneficial to Maple, and thus should not be overlooked for the sake of compatibility with the older machines. Thus, the IBM PC port will probably be divided into three separate projects.

A Maple Machine based port will be performed in order to allow Maple to be run on the 8088/8086 based machines. This Maple Machine will run under the PC-DOS operating system, making use of the 640 kilobytes available. The results of this port will also run on the newer 80286 and 80386 machines, but will not take advantage of any of their features.

Another Maple Machine based port will be performed specifically for 80286 based machines. The resulting Maple Machine will run under PC-DOS, OS/2, and possibly AIX. Maple Machines on 80286 based hardware will be able to access up to 16 megabytes of memory. This will require a special utility program to function under PC-DOS, since that operating system limits memory accesses to the first 1 megabyte of the 80286 address space by running the CPU in "real" mode.

Finally, a port will be made to 80386 based machines. These machines can emulate the 8086, or they can run in native mode, where they are true 32-bit machines. The address space is still segmented, but each segment can be 4 gigabytes in length. Thus, the machine effectively has a linear address space. Since this is a 32-bit machine with a linear address space, it should be possible to compile Maple into 80386 machine code. Such an implementation will probably only function under AIX, although it may be possible to create a DOS version using one of the commercial DOS extender packages.

9.2. New Instructions

The Maple Machine currently supports an instruction set of 95 instructions. This leaves 33 unused opcodes out of the maximum of 128 possible codes. Although little will be gained by adding new instructions to replace common pairs, nothing would be lost. And by creating new instructions to perform even more complex operations, some substantial gains might be made. This is especially important for the IBM PC port, since the target machine is quite slow to begin with.

9.3. A Threaded Interpreter

The current implementations of the interpreter all employ a central loop, wherein the next instruction is fetched at the top of the loop, and the appropriate code is branched to based on the instruction opcode. After executing the instruction, control must be transferred back to the top of the loop to execute the next instruction.

If the interpreter is to be coded in assembly language, an alternate approach suggests itself. At the cost of additional space requirements, code can be added to the end of the execution routine for each instruction to fetch the next instruction and branch to the desired location. Thus, control transfers from one execution routine to the next, like beads on a thread, and is known as threaded code [1]. It is actually indirect threaded code [5], since the instructions are not the actual addresses of the execution routine, but rather are used as an index into an address table. A partially indirect threaded code technique is also possible, in which the address of an instruction's execution routine can be computed from the instruction opcode.

The threaded code technique has the advantage of not requiring a branch (and the associated overhead) back to the controlling part of a main interpretation loop. The extra code required in each execution routine would not be very large. Even if this code was 16 bytes, this would only increase the size of the interpreter by 2 kilobytes if there were 128 instructions.

9.4. Global Optimization

As was discussed in Chapter 7, the Maple Machine compiler does not perform any global optimization. The Maple source code has already been extensively hand optimized over the years, and it is only recently that compilers have been able to improve it significantly. If such a compiler were available for the Maple Machine, performance might be enhanced.

9.5. Merging the Interpreter and the Maple Kernel

Some parts of the Maple kernel are executed quite frequently, and are responsible for a large portion of the time that Maple spends executing. Some of these parts are fairly low level, and not directly involved with the business of symbolic computation. The most prominent of these are the storage allocation and garbage collection routines.

It might be feasible to extract some of these low level routines from the Maple source code, and provide their capabilities as instructions in the Maple Machine. The C compiler could then translate calls to these routines into single instructions. All the instructions that were formerly executed each time one of these routines was called would then be replaced by a single instruction, whose operations are executed by the underlying hardware at proportionally higher speeds. This would of course make the Maple machine less portable.

Of course, one could get carried away and implement more and more of the Maple kernel as part of the Machine. In the end one would have a Maple Machine with only one instruction, with the mnemonic MAPLE. This would effectively be a traditional port.

9.6. A Hardware Maple Machine?

The author is a firm believer in throwing silicon and solder at a problem, and has often entertained the idea of constructing a Maple Machine in hardware. Such a machine would of course not contribute to the portability of Maple (unless the machine weighed very little), but would probably result in one of the fastest Maple implementations ever achieved.

References

[1] Bell, J.R., "Threaded Code", *Communications of the ACM*, **16**, 370-372, (1973).

[2] Bell Telephone Laboratories, Inc., *UNIX Time-Sharing System, UNIX Programmer's Manual*, Holt, Rinehart and Winston (1983).

[3] Char, B.W., Geddes, K.O., Gonnet, G.H., Monagan, M.B., Watt, S.M., *MAPLE Reference Manual, 5th Edition*, WATCOM Publications Limited, Waterloo, Ontario (1985).

[4] Cheriton, D.R., Malcom, M.A., Melen, L.S., Sager, G.R., "Thoth, a Portable Real-Time Operating System", *Communications of the ACM*, **22**, 105-115 (1979).

[5] Dewar, R.B.K., "Indirect Threaded Code", *Communications of the ACM*, **18**, 330-331, (1975).

[6] Eberle, H., *Hardware Description of the Workstation Ceres*, Institute für Informatik, Eidgenössische Technische Hochschule Zürich, (1987).

[7] Gardner, J.A. *A Tutorial Guide to the Language B*, University of Waterloo, Waterloo, Ontario, Canada (1980).

[8] Hendrix, J.E., *The Small-C Handbook*, Reston Publishing Company, Inc., Reston Virginia (1984).

[9] Intel Corporation, *Microprocessor and Peripheral Handbook*, Intel Corporation, Santa Clara, California (1983).

[10] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Bell Laboratories, Murray Hill, New Jersey (1978).

[11] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software – Practice and Experience*, **1**, 105-173, (1971).

[12] Lecarme, O., "Pascal and Portability", *Pascal – The Language and its Implementation*, 21-35, John Wiley & Sons, Ltd., Chichester – New York – Brisbane – Toronto (1981).

[13] Miller, D.L., "Stack Machine and Compiler Design", *Byte — The Small Systems Journal*, Vol.**12**, No.4, 177-186 (1987).

[14] Moore, C.H., Rather, E.D., *FORTH — An Application-Oriented Language Programmer's Guide*, National Radio Astronomy Observatory, Green Bank, W. Virginia.

[15] Nori, K.V., Ammann, U., Jensen, K., Nageli, H.H., Jacobi, Ch., "Pascal-P Implementation Notes", *Pascal — The Language and its Implementation*, 125-170, (1981).

[16] Ohran, R., "Lilith and Modula-2", *Byte — The Small Systems Journal*, Vol.**9**, No.8, 181-194 (1984).

[17] Poole, P.C., Waite, W.M., *Advanced Course in Software Engineering*, Springer-Verlag, Berlin — Heidelberg — New York (1973).

[18] Richards, M., *BCPL, The Language and its Compiler*, Cambridge University Press, Cambridge — New York (1979).

[19] Ryder, B.G., "The PFORT Verifier", *Software — Practice and Experience*, **4**, 359-377, (1974).

[20] Tannenbaum, A.S., "Implications of Structured Programming for Machine Architecture", *Communications of the ACM*, **21**, 237-246 (1978).

[21] University of California, *UNIX Programmer's Manual: 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, University of California, Berkeley (1983).

[22] Welsh, J., Sneeringer, W.J., Hoare, C.A.R., "Ambiguities and Insecurities in Pascal", *Pascal — The Language and its Implementation*, 5-19, (1981).

[23] Wirth, N., "Pascal-S: A Subset and its Implementation", *Pascal — The Language and its Implementation*, 199-259, (1981).

[24] Wirth, N., *Programming in Modula-2*, Springer-Verlag, Berlin — Heidelberg — New York — Tokyo, (1985).

[25] Wirth, N., Gutknecht, J., *The Oberon System*, Institute für Informatik, Eidgenössische Technische Hochschule Zürich, (1988).

[26] Wirth, N., "The Programming Language Oberon", *Software — Practice and Experience*, **18**, 671-690 (1988).

[27] Wirth, N., "From Modula to Oberon", *Software — Practice and Experience*, **18**, 661-670 (1980).

Related Works

Bal, H.E., Tannenbaum, A.S., "Langauge- and Machine-Independent Global Optimization on Intermediate Code", *Computer Language*, **11**, 105-121 (1986).

Berry, R.E., "Experience with the Pascal P-Compiler", *Software — Practice and Experience,* **8**, 617-627 (1978).

Coleman, S.S., Poole, P.C., Waite, W.M., "The Mobile Programming System, Janus", *Software — Practice and Experience*, **4**, 5-23 (1974).

Davidson, J.W., Gresh, J.V., "Cint: A RISC Interpreter for the C Programming Language", *SIGPLAN Notices*, Vol.22, No.7, (1987).

Klint, P., "Interpretation Techniques", *Software — Practice and Experience*, **11**, 963-973 (1981).

Newey, M.C., Poole, P.C., Waite, W.M., "Abstract Machine Modelling to Produce Portable Software — A Review and Evaluation", *Software — Practice and Experience*, **2**, 107-136 (1972).

Richards, M., "The Portability of the BCPL Compiler", *Software — Practice and Experience*, **1**, 135-146 (1971).

Tannenbaum, A.S., van Stavern, H., Keizer, E.G., Stevenson, J.W., "A Practical Tool Kit for Making Portable Compilers", *Communications of the ACM*, **26**, 654-660 (1983).

Tannenbaum, A.S., van Stavern, H., Keizer, E.G., Stevenson, J.W., *Description of a Machine Architecture for use with Block Structured Languages*, Informatica Rapport IR-81, Vrije Universiteit, Amsterdam (1983).

Tannenbaum, A.S., van Stavern, H., Stevenson, J.W., "Using Peephole Optimization on Intermediate Code", *ACM Transactions on Programming Languages and Systems*, **4**, 21-36 (1982).

Waite, W.M., "The Mobile Programming System: STAGE2", *Communications of the ACM*, **13**, 415-421 (1970).