A C Compiler for a Capability-Based Processor

Roland Yap, John Rosenberg, and David Abramson

Department of Computer Science
Monash University
Clayton, Victoria 3168

ABSTRACT

This paper describes the development of a C compiler for MONADS-PC, a capability-based workstation designed and implemented at Monash University. The MONADS architecture is briefly described and this is followed by a discussion of the problems associated with mapping C onto a capability-based environment. The implementation of the compiler using the Amsterdam Compiler Kit (ACK) is described and the paper concludes with a discussion of some of the problems encountered with ACK.

Keywords and phrases: Compilers, C, capabilities, protection.

CR categories: C.1.3, D.3.4, D.4.2.

1. INTRODUCTION

The MONADS-PC computer system [Rosenberg and Abramson, 1985] was developed at Monash University during 1984 as a workstation to support software engineering research. The 32 bit microprogrammed processor provides hardware support for the decomposition of complex software systems into information-hiding modules [Parnas, 1972]. The main features of the machine are:

1. A large (2^{60} bytes) uniform virtual memory used to hold both computational data and permanent data (i.e. files).
2. A segmented address space.
3. A capability-based protection mechanism.

Due to the large amount of general-purpose utility software available under the UNIX operating system and written in C (and the lack of such software for MONADS-PC), it was desirable to develop a C compiler for the MONADS-PC. Block structured and strongly-typed languages in the style of Pascal [Jensen and Wirth, 1975] and Modula 2 [Wirth, 1980] map easily onto the MONADS architecture. However, during the design of the new compiler several difficulties were found with mapping C onto MONADS.

This paper describes the essential features of the MONADS architecture and discusses the conflicts between the C environment and the MONADS philosophy. The proposed design and implementation details are presented.

2. OVERVIEW OF THE MONADS ARCHITECTURE

The main features of the MONADS architecture are the large addresses required to access the virtual memory, capability-based protection and the procedure calling mechanism. In this section we will briefly describe how these are implemented and how they may be used by languages such as Pascal.

2.1. Addressing Structure

The idea of capability-based addressing was first proposed by Dennis and Van Horn [1966] and can be summarised as follows. Every object in the system is given a unique, non-forgeable address called a capability. These addresses are never re-used, even when the object has been deleted. Capabilities themselves are protected and cannot be modified or directly generated by programs, rather they are created and transferred under system control. A particular object can only be accessed if a procedure has a capability for the object. Thus a fine grain of protection and control over access to data can be achieved. The MONADS architecture exploits this property of capabilities in order to implement information-hiding modules.

MONADS virtual addresses are 60 bits in size. The address is divided into two parts, an address space number (32 bits) and an offset (28 bits). An address space is used to hold a related group of segments (e.g. the code of a module, the data of a module or a stack). When an address space is created it is allocated a unique number and this number is never re-used, even after the address space has been deleted. Address spaces are paged, with a page size of 4096 bytes. The 28 bit offset portion of a virtual address identifies a byte within the address space.

Each address space is divided into a number of segments. Segments are not related to pages or page boundaries. Programs never directly use virtual addresses, rather they address segments via segment lists. A segment list contains a number of capabilities for segments in an address space. Thus a segment list entry is effectively a capability for a window on the address space. The entire structure is illustrated in figure 1.

Each process in MONADS has a stack address space. This is used to hold linkage information on procedure calls, local data for procedures and for temporary storage during expression

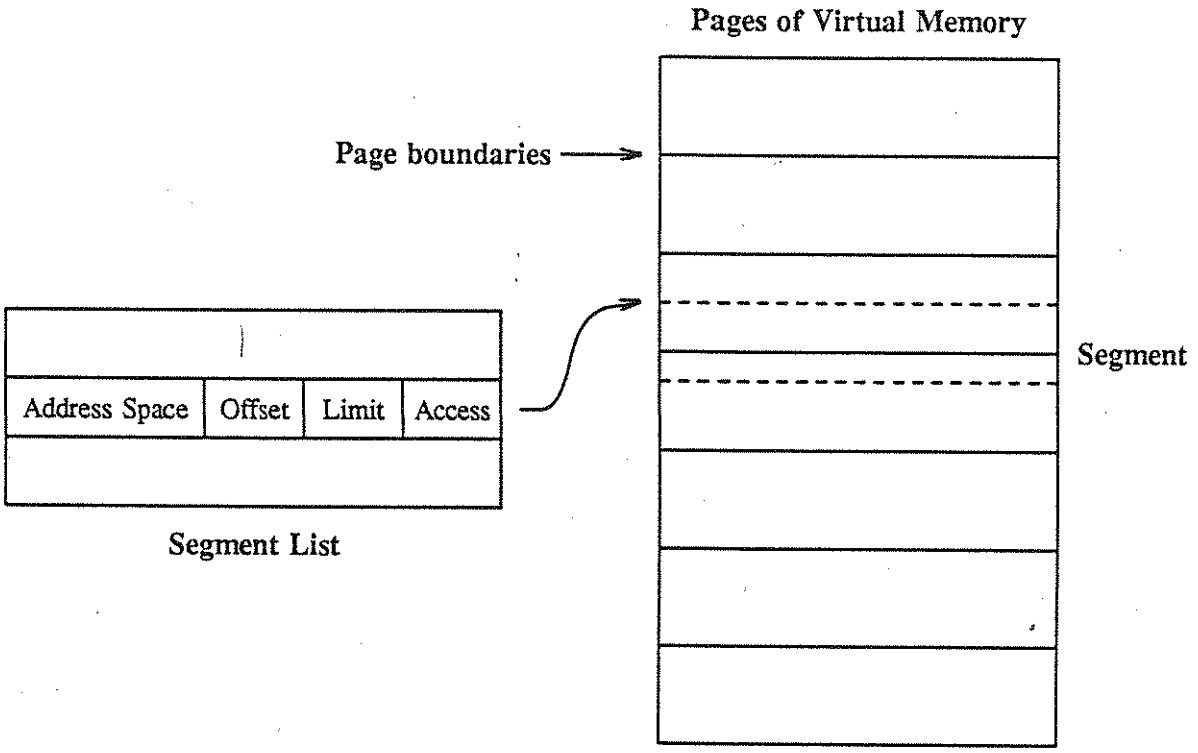


Figure 1 - MONADS Addressing Structure

evaluation. The data on the stack is also held in segments and is addressed via segment lists, themselves held on the stack.

The addressing environment of a process on MONADS is defined by the segment list entries of the segment lists which it can use. The valid segment lists for a process are pointed to by a table held at the base of the stack, in an area of memory not directly addressable by programs. The entries in this table are called *bases*, and point at segment lists for the data segments which are currently addressable by the process [Keedy, 1982]. Access to a byte in the virtual memory is achieved by specifying a base, segment list entry and an offset within the segment. Naturally, the bases and segment list entries are only modified by special system management instructions.

The MONADS architecture supports up to 64 bases and 1024 segments per segment list. Thus 16 bits are needed to identify uniquely a segment list entry. In order to reduce the size of addresses in instructions, and for efficiency reasons, a set of 16 capability registers are provided. These have the same format as segment list entries and may be loaded from a segment list by a system management instruction. Thus machine instructions can address data by specifying a capability register and an offset.

From the above, it can be seen that the addressing environment at any time is defined by the segment lists pointed to by the bases. This provides very flexible control over access to data. On a procedure call, the call mechanism modifies the bases to create the appropriate environment for the called procedure. This environment is derived from "red-tape" information held in non-addressable segments in the address space(s) of the called module. Information from this red-tape is also used to create local data segments on the stack for the called procedure. The call mechanism also performs several other useful tasks and these are fully described in [Keedy, 1982].

2.2. Instruction Set

MONADS-PC is a single accumulator machine [Rosenberg, 1984]. The accumulator is an implicit operand on most arithmetic and logical operations. The accumulator (A) is a 32 bit register. There is also an accumulator extend register (AX) which can be used for double length

arithmetic operations. There are three index registers (I1-I3) and sixteen capability registers (C0-C15) of the format shown earlier. In addition there are two dedicated capability/index register combinations, one for accessing the computational area of the stack and one for fetching words of code from the instruction stream.

MONADS-PC has three basic addressing modes, data mode, literal mode and top-of-stack mode. In data mode a capability register, offset and optionally an index register are specified. The capability register effectively points to a window on the virtual address space, the offset is an offset from the start of this window and the index register is added to this offset. Both the offset and the index register must be positive. Thus data mode can be used to access any byte in the addressing environment of the current process. Literal mode is used to generate a constant literal. Top-of-stack mode provides access to a computational (push-pop) area of the stack. This area *cannot* be accessed via a capability register and the hardware guarantees that a pop operation will not cause the top-of-stack pointer to go below the start of the computational area. Furthermore the top-of-stack pointer cannot be directly manipulated by a program.

2.3. An Example: Pascal

This section demonstrates how the architecture is used to implement block-structured languages such as Pascal. Figure 2 is a sample program in which there is some scalar global data (i, j, k) and a global array (a). The program has one procedure declared at lexical level one. This procedure is passed two parameters, ref_param by reference and val_param by value, and has one local variable (m).

Figure 3 shows the stack structure immediately after the procedure call. At the bottom of the stack is the base table for the process. This table contains (protected) pointers to the three segment lists currently addressable by the program. On entry to the program a segment list for global (lexical level zero) data was created. This contains two entries, one for the global scalars (by

```
program example;
type
        array_type = array[0..9] of integer;
var
        i, j, k   : integer;
        a         : array_type;

procedure local_proc(var ref_param : array_type;
                         val_param : integer);
var
        m         : integer;
begin {local_proc}
        .....
        .....
end {local_proc};

begin {mainline}
        local_proc(a, 5);
        .....
        .....
end {mainline}.
```

Figure 2 - A Sample Pascal Program

convention all scalars at the same lexical level are packed into a single segment by the compiler) and one for the array (a). The segment list is followed by the actual data segments.

Execution of the procedure call to local_proc involves the creation of a parameter segment list of length two, a value parameter segment to hold the constant 5, the call linkage (containing amongst other things the return address), a segment list for local data at lexical level one and a data segment for the local variable m. The first entry in the parameter segment list points at the reference parameter a (ref_param) and the second entry points at the value parameter segment created above it. CAB is the Computational Area Base register and points to the start of the push-pop area. TOS is the Top-of-Stack register mentioned earlier.

The program gains access to a data segment by loading a capability register from a segment list pointed to by one of the valid bases. This, in conjunction with push and pop, is the only method of accessing data. The bases, segment lists and call linkage are created and manipulated only by special system management instructions. The integrity of the system depends on this restriction.

One of the major advantages of the segment structure is that it provides hardware bounds checking on all data structures. Unfortunately this is also one of the major difficulties with implementing C on MONADS-PC.

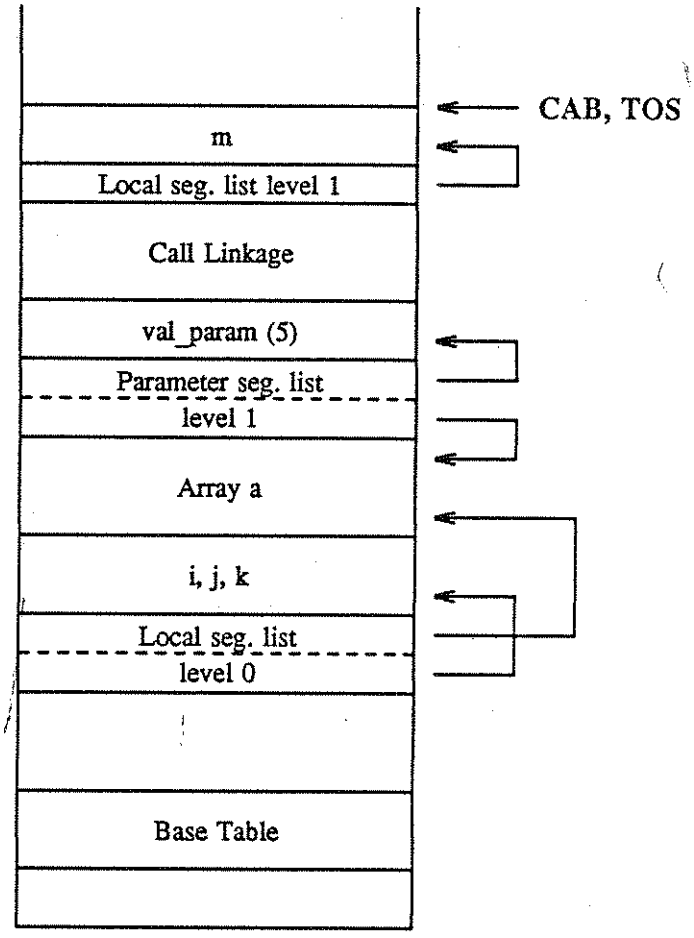


Figure 3 - Stack Structure After Pascal Call

3. THE C PROGRAMMING LANGUAGE AND ENVIRONMENT

A brief description of important features of C [Kernighan and Ritchie, 1978] pertaining to compiler implementation is given here. C is a block structured high level language with certain low level machine-like features and relatively weak typing. The fundamental data types are integers and floating point numbers of various sizes, characters and pointers. New types can be derived from the basic data types using arrays, structures (similar to pascal records) and unions (similar to pascal variant records). All data objects can be initialised at compile time.

Data objects may be referenced indirectly by using a pointer. Any data object of type *t* may be addressed via a pointer of type *pointer-to-t*. Using a *cast* operator the same pointer may be coerced to address a data object of a different type. Unlike Pascal, C also provides pointers to functions and thus a function may be called using a pointer. Pointer arithmetic is "well-defined". Pointers are implicitly allowed to address the whole of the data space of the executing program. Arrays in C are also very closely related to pointers. In fact accessing a variable via an array reference or a pointer reference is virtually identical.

A C program consists of declarations and functions. Unlike Pascal C does not allow nested function declarations. All functions are declared at lexical level 1 and therefore only have access to local and external (global) data. Argument passing is call by value only, however call by reference can be achieved by passing pointers (addresses). C is unusual in that it allows functions to be called with a variable number of parameters. The called function does not have anyway of knowing how many arguments are passed to it except by convention. The two most common ways are by specifying the number of parameters in the first argument or by terminating the argument list by some special value, say zero. Access to the variable arguments is usually achieved via a machine dependent library routine. C allows "goto's" but these are restricted to localized jumps within a function. Jumping across functions (which usually involves collapsing of stack frames) is achieved with the library routines *setjmp* and *longjmp*.

4. MAPPING C ONTO THE MONADS ARCHITECTURE

The nature of C conflicts with the fundamental philosophy of MONADS-PC. The main problem concerns the use of pointers in many different manifestations. These problems are not unique to the MONADS architecture but are inherent in other capability-based or segmented architectures (eg. 8086 microprocessor).

4.1. Problems with Pointers

Most C compilers implement pointers by using machine addresses and thus a pointer may be used to address any region of memory within the address space of the process. Similarly pointers may be modified in an arbitrary manner (e.g. to step through elements of an array). Thus pointers are simply an alternative (indirect) path for access to data.

Such an approach is not possible on MONADS. Processes are not permitted to generate virtual addresses directly because this would allow a process to violate the integrity of the system. The only method of access to data on MONADS is via capability registers pointing at segments. This suggests that pointers on MONADS should be implemented as capabilities. However, this is not feasible for two reasons. First, the capabilities would have to be protected by being placed in separate capability segments. This segregation would complicate management of structures containing both pointer and other data. The second reason for not implementing pointers as capabilities is that pointers can be modified arbitrarily, whereas capabilities cannot. It may be possible to provide special instructions allowing controlled modification of capabilities, but then the question of efficiency must be considered. C programs tend to make use of pointers heavily because it is usually the most efficient coding style on machines like a VAX, especially when register pointer variables are employed. It is therefore desirable to try to ensure that pointer accessing and modification is reasonably efficient.

The solution adopted is to create a single contiguous segment for the data space of a C program, where a pointer is just an offset into the (well-defined) segment. Thus, pointers may be modified arbitrarily without compromising system security because all that can be accessed is data within that segment. If multiple segments were to be used (e.g. one for each data structure) then a capability for a segment would have to be associated with each pointer and for reasons described above this is not permissable. A major disadvantage of the single-segment scheme is that all of the advantages of the hardware support for capabilities and protection *within* C programs are lost. However, this solution has the advantage of being simple and efficient.

4.2. Dynamic Storage Allocation

While C does not have any intrinsic dynamic allocation, C programs usually make extensive use of library routines to do so (e.g. *malloc*). It is desirable on a segmented architecture to have a separate segment for the heap, thus providing a truly dynamic heap which can grow and shrink independently from the stack. Previously it was shown that the C environment should be contained in a single segment. This limits us to having the local data and the heap in the same segment, and thus the same address space. Because the MONADS virtual address space is very large, allocating the whole of the possible address space will consume some additional overhead in page tables. It is therefore proposed that the C environment will consist of a single large contiguous data segment whose size can be specified at run-time. This segment can actually be created on the normal process stack. The value of pointers or addresses will then be offsets from some specified base capability for this segment.

4.3. External/Global variables

C supports the initialisation of variables at compile time. Initialisation of local variables is normally achieved by run-time code executed on entry to the relevant function and thus presents no problem. Global or external variables, however, have to be initialised at compile time. When a program is executed on a conventional computer, a new copy of the code and data is loaded into memory from disk. However, on MONADS-PC a compiled program persists in the uniform virtual memory between invocations. Thus any global data which has been modified by a previous invocation must be re-initialised. Since the environment is actually created on the process stack at run-time global data is best supported by copying initial values for the global data from a constant (read-only) data segment on entry to the program. This is analogous to a more conventional execution of a program, say on UNIX, where the data segment is loaded from a file.

5. IMPLEMENTATION

Two "portable" C compilers were available to implement the MONADS-PC C compiler, the Amsterdam Compiler Kit (ACK) [Tanenbaum et al, 1983; Li and Schwetman, 1984] and the Portable C Compiler (PCC) [Johnson, 1977]. The Amsterdam Compiler Kit was selected for a number of reasons. Transporting PCC to a new machine usually involves writing 1000-2000 lines of code and requires intimate knowledge of the compiler source. On the other hand, ACK is a higher level tool where only the code generator needs to be produced from a high level description of the translation process. Thus the choice was between a data-driven method (ACK) and a more procedural approach (PCC) and it was felt that the table-driven method would be more flexible and ease maintenance. ACK is also a more complete development environment which includes other language front-ends (eg. Pascal and Basic) as well as optimizers and a portable assembler.

5.1. Description of the Amsterdam Compiler Kit (ACK)

The Amsterdam Compiler kit provides a number of independent language front-ends which compile into code for a virtual stack machine, called EM, which is similar to P-Code [Ammann, 1977] but more general. The task of the compiler writer is to develop a code generator for the particular machine to translate from EM into native machine instructions. The code generator (cg)

is automatically produced by a code generator generator (cgg) from a high level description. This description is in the form of rules describing the many-to-many mapping from sequences of EM instructions to machine instructions. The code generator works by simulating a *fake* stack for the EM machine stack and using the rules to cause appropriate actions. These actions include emitting strings (assembler output), substituting EM sequences with other EM instructions, manipulating the fake stack, performing book-keeping such as register allocation and keeping track of movement of data objects. When a function call or conditional/unconditional branch is encountered the fake stack has to be emptied as the structure of the stack following the call/branch is not known. The clearing of the fake stack is also necessary to allow for parameter passing. Special rules are required to map the fake stack onto the real machine.

5.2. Stack Frame Layout

The Amsterdam Compiler Kit places some restrictions on how locals and parameters are addressed. It requires that a local base (see figure 4) is set up properly on a function call, such that, relative to the base, parameters are addressed with positive offsets and locals are addressed with negative offsets (assuming downward growing stack frames). This is similar to the scheme used by compilers on the VAX, except that the VAX has an additional register (argument pointer) to address parameters. Locals are addressed as negative offsets from the frame pointer (local base). The advantage of using both negative and positive offsets is that the number of parameters need not be known to the callee.

This scheme has to be modified as MONADS-PC does not allow negative offsets (specifying negative offsets to capabilities is not meaningful). Since the number of parameters is unknown to the callee, it is still desirable to use positive offsets relative to a dedicated capability register which points to the arguments. The number of locals required is known at compile time. By reversing the above scheme, locals can be addressed as positive offsets from a local base (capability register) which points to the bottom rather than the top of the stack frame. Thus locals run from low to high addresses.

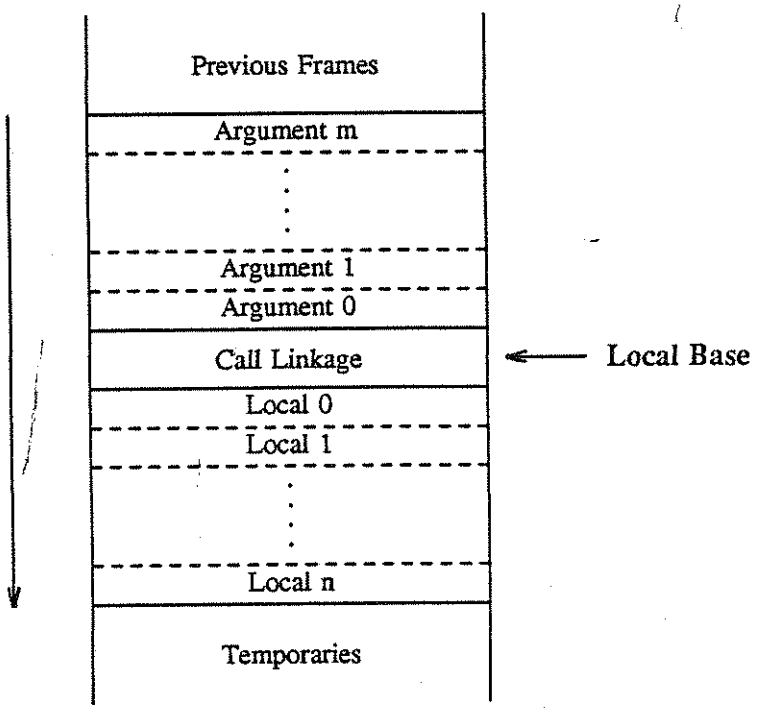


Figure 4 - ACK Stack Structure

The stack frames cannot be placed on the real hardware stack because the real stack can only be manipulated through pushes and pops. Therefore a software stack has to be simulated in the program data space. Because of the method of addressing of locals and parameters the stack should grow downwards. In fact both the real stack and a simulated stack are used. The real hardware stack is used to store temporaries in evaluating expressions and linkage information, whereas the simulated one is for local variables and passing parameters.

Altogether four capability registers are used to provide the C run-time environment (see figure 5). In a stack frame, the *local base* (LB) points to local data for the function and the *argument pointer* (AP) points at the first argument. Global data is accessed via the *external base* (EB) and the *global base* (GB) points to the bottom of the whole data segment. Pointers are addressed as offsets from the global base. An index register is chosen to be the simulated stack pointer (SP). Operations on the simulated stack are achieved by indexing using SP relative to the global base.

Some new (microcoded) instructions have been provided to manipulate these four capability registers safely (i.e. these instructions will not violate system integrity). The same instructions also support non-local jumps and stack unwinding.

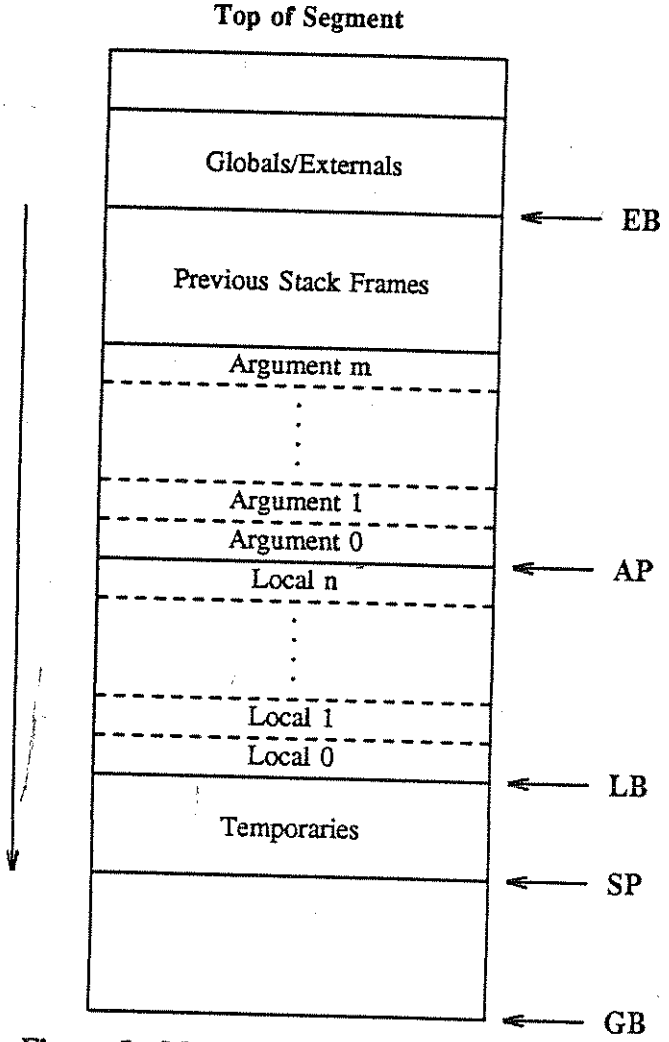


Figure 5 - Mapping EM onto MONADS

5.3. Writing the Code Generator

Writing the code generator consists mainly of writing rules in the following form:

Rule = EM pattern '|' stack pattern '|' actions '|' stack replacement '|' EM replacement '|' [cost]

where the EM pattern can contain conditional expressions and the stack pattern is that of the fake stack. A rule is triggered when the EM code and the fake stack match the patterns in the rule. More than one rule can match and the code generator chooses the rule with lowest cost. Since MONADS-PC is a single accumulator machine, not all operations are supported on index registers, nor can arithmetic operations be performed between the accumulator and index registers. Particular emphasis was placed on generating efficient code for expression evaluation (including assignment, accessing structures, arrays, pointers etc.), which is the biggest component of the compiler. The rest of the rules are concerned with control flow.

It proved difficult to write the rules to eliminate redundant operations. The difficulty arises because EM is a stack machine and temporaries created during expression evaluation are kept on the stack. On a multi-register machine this problem does not arise as one simply allocates new registers to store the temporaries (until the registers spill over). The same approach would not work on MONADS-PC since there is only one register and thus a lot of stacking of temporaries would occur. Due to the way ACK manages its fake stack, such temporaries would be placed on the software stack (requiring two instructions) instead of the hardware stack. A more complicated approach to keeping track of the accumulator in the fake stack was tried, however, it was found that the rules to do this simply grew out of hand so another approach was used.

The code generator produces "naive" code (ie. a lot of "pushes" and "pops") when evaluating an expression. The resultant code is then passed through a peep-hole optimizer which removes redundant instructions and replaces instruction sequences with more efficient ones. The hardware stack is also used to store temporaries. As a result, we ended up with considerably simpler rules for the code generator, making it easier to write and understand and at the same time improving efficiency.

5.4. Problems with the Amsterdam Compiler Kit

Quite a number of problems were encountered when using ACK. The documentation is rather poor, only briefly describing the features of the kit. A number of bugs were also discovered. Bugs are difficult to fix as program documentation varies from being sparse to non-existent. The sample code generators provided do not describe why the rules were written in the particular fashion, nor are they very helpful when trying to solve problems different from the examples. Admittedly the sample code generators would be very useful if one were implementing a code generator for a similar machine, but they were not of much use to us as our machine was quite different.

The code generator generator is also not particularly well written. It produces a C program mainly consisting of tables. However insufficient semantic checking is performed and the resultant program may not even compile. Similarly errors in the rules can cause the code generator to crash displaying a cryptic message. This message does not relate the fault to the original code generation rules, making testing and debugging difficult.

6. CONCLUSION

The MONADS-PC C compiler is partially complete. A peephole optimizer has been written and is used on the output from the code generator. Significant improvements have been obtained and the quality of the final code is comparable to hand written assembler code.

MONADS-PC runs its own operating system above a kernel which supports the uniform virtual memory and basic input-output management. An area of interest in the future is the transporting of the UNIX operating system to run on MONADS-PC. It is anticipated that UNIX would run

above the existing kernel in parallel with the MONADS system.

The major conclusion that can be drawn from our experiences with this project is that languages such as C do not map cleanly onto highly structured architectures such as MONADS. This is because these languages were developed with the implicit assumption that they would be executed on a traditional Von Neumann machine. In order to fully and efficiently exploit the advantages of capability-based architectures new languages will be required. Some work in the language design area is being undertaken as part of the MONADS project [Evered, 1985].

REFERENCES

Ammann, U. (1977): "On Code Generation in a Pascal Compiler", *Software Practice and Experience*, 7, 3, pp. 391-423.

Dennis, J.B. and Van Horn, E.C. (1966): "Programming Semantics for Multiprogrammed Computations", *Comm. ACM*, 9,3, pp 143-155.

Evered, M. (1985): "LEIBNIZ: A Language to Support Siftware Engineering", Dr. Ing., Technical University of Darmstadt, 1985.

Jensen, K. and Wirth, N. (1975): "Pascal User Manual and Report", Springer-Verlag.

Johnson, S. C. (1977): "A Tour Through the Portable C Compiler", UNIX Programmers Manual Vol. 2.

Keedy, J.L. (1982): "Support for Software Engineering in the MONADS Computer Architecture", Ph.D. Thesis, Department of Computer Science, Monash University.

Kernighan, B.W. and Ritche, D. M. (1978): "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey.

Li, K. and Schwetman, H. (1984): "Implementing a Scalar C Compiler on the Cyber 205", *Software Practice and Experience*, 14, 9, pp. 867-888.

Parnas, D.L. (1972): "On the Criteria to be Used in Decomposing Systems into Modules", *Comm. ACM*, 15, 12, pp 1053-1058.

Rosenberg, J. (1984): "MONADS-PC Instruction Set", MONADS-PC Technical Report 1, Department of Computer Science, Monash University.

Rosenberg, J. and Abramson, D. (1985): "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", *Proc. 18th. Annual Hawaii International Conference on System Sciences*, Honolulu.

Tanenbaum, A. S., Steveren, H. V., Keizer, E. G. and Stevenson, J. W. (1983): "A Practical Tool Kit for Making Portable Compilers", *Comm. ACM*, 26, 13, pp. 654-660.

Wirth, N. (1980): "Modula-2", E.T.H., Zurich.

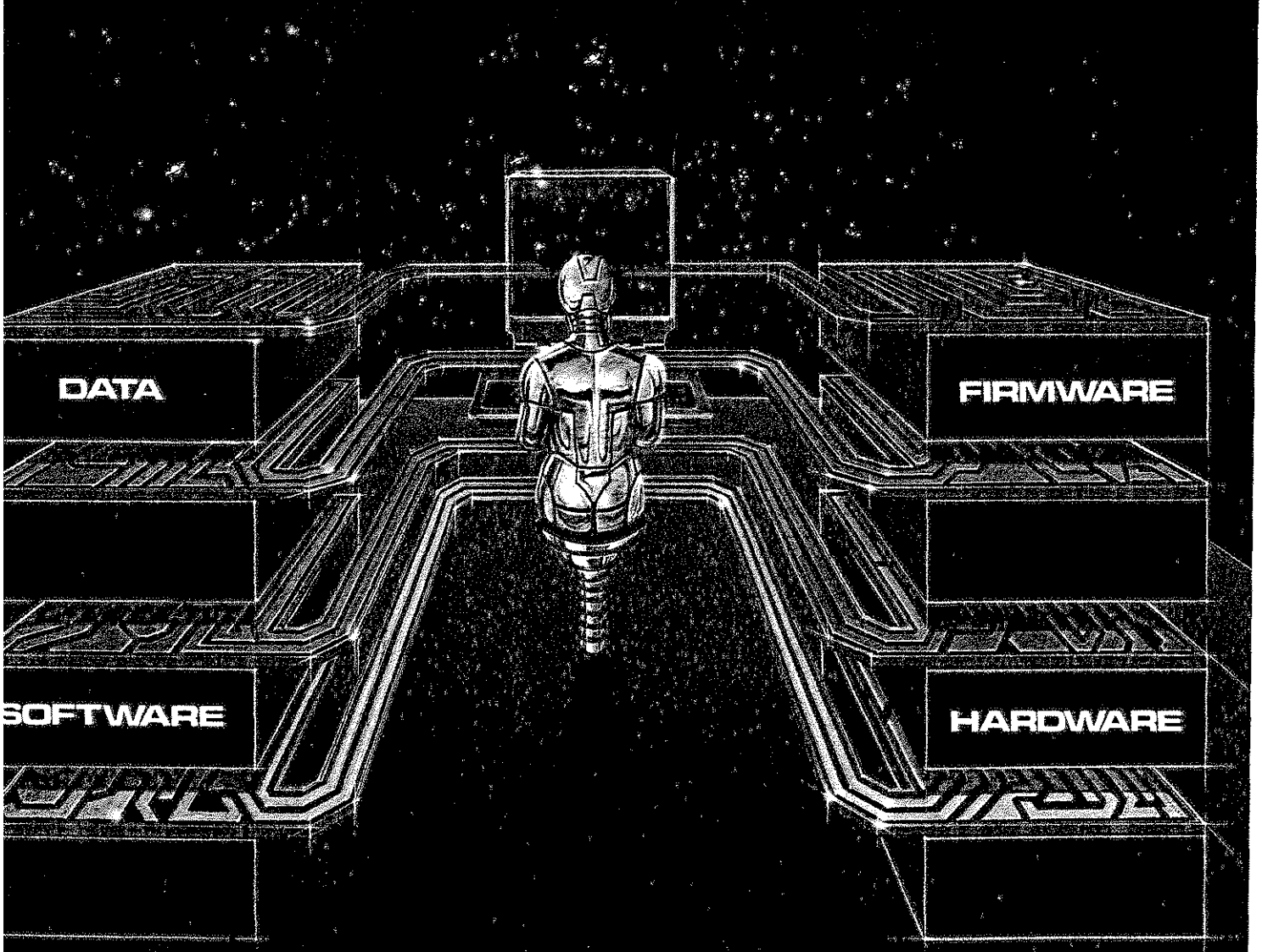
9th AUSTRALIAN COMPUTER SCIENCE CONFERENCE

Canberra 1986

ANU

UNIVERSITY COLLEGE

JANUARY 29—31



APHIC PROVIDED BY;
VANCED TECHNOLOGY CENTRE
PARTMENT OF INDUSTRIAL DEVELOPMENT
D DECENTRALISATION, NEW SOUTH WALES.

PROCEEDINGS SPONSORED BY;
APPLIED RESEARCH & DEVELOPMENT DIVISION

COMPUTER POWER