A NEW MODULA COMPILER FOR THE LSI-11

Ahmed Mahjoub
Philips Laboratories
345 Scarborough Road,
Briarcliff Manor, NY 10510

Abstract

This note contains a brief description of a new Modula compiler developed at Philips Laboratories. The compiler generates code for the LSI-11 microprocessor. It is written in Pascal and operates under control of the U.C.S.D. system. Philips Laboratories Modula (PL Modula) differs slightly from original Modula as defined by Wirth. These differences are outlined in section 3.

1. Introduction

PL Modula is a slightly changed and extended version of the language Modula defined by N. Wirth [1,2,3]. The most notable changes and/or extensions are: a priority mechanism for regular processes, structured constants and variant records.

As part of the ongoing higher level language activities at Philips Laboratories a two-pass compiler for PL Modula has been developed. The compiler is currently operational, and runs on the TERAK personal computer (LSI-11-based with 64k bytes of RAM).

To enhance the portability of the software developed in PL Modula, an intermediate language (P-code) is used. The compiler consists of two distinct parts: A translator that accepts PL Modula source programs and produces equivalent P-code programs, and a code generator that accepts a P-code programs and produces stand alone load modules.

2. Modula

Modula is a Pascal-based structured language that supports concurrent programming. Modula has two of the principal features of Pascal: strong typing and user defined types. Its main structuring tools are the process and the module. A process consists of private and shared data and is expected to execute concurrently with other processes. A module is a collection of objects (e.g. variables, procedures, types, etc...) which are local to the module, but which persist throughout invocations of its procedures. The main purpose of the module strucuture is to hide the data it contains and provide explicit control on

---

Editor's Note: The compiler is intended for non-commercial use, and can be made available under license agreement to university research facilities for research purposes at $150.00 per copy. For more information, write to: Modula Project, Computer Systems Research Group, at the address on top of this page.

outside access to that data.

In addition to regular modules, Modula has two other kinds of Modules: Interface Modules, which are very similar to monitors in that they are used for communication and synchronization of processes, and device modules, which are interface modules that can contain machine dependent code. These modules are purposely designed for implementing device drivers and I/O routines.

One of the principal aims in the design of Modula is to be able to run programs on a bare machine with minimal run-time support. Current implementations have shown that this is indeed possible and the size of the kernel has been reported as 100-150 words [2,4]. This makes Modula programs execute efficiently and makes the language especially suited for real-time applications and low level device handling.

3. Some differences with the original definition

The most significant extension we made to Modula is the definition of a uniform priority mechanism. We allow regular processes to have non-zero priorities, but restrict these priorities to be lower than those of device processes. We also allow device processes to wait on signals emitted by other device processes. The implications of this extension with regard to the language implementation are as follows:

1) The ring implementation specified in [2] is no longer valid. Instead a priority queue is used. All ready processes are linked into a ready queue and scheduled on a priority basis. Context switching occurs according to the following rule: Let P1 and P2 be two (regular or device) processes having priorities i and j respectively. Assume P1 sends a signal awaited by P2, then

i>j The sending process continues. The waiting process is appropriately inserted in the ready queue. There is no context switch.

i<j The sending process releases the processor and is appropriately inserted in the ready queue. The signalled process moves to the head of the queue and receives control of the processor. In this case there is a context switch.

i=j If the sending process is a device process then case i>j is adopted else case i<j is adopted. This is consistent with Wirth's goal of minimizing context switches inside device modules so that devices can run more efficiently.

A context switch may also occur when a regular process exits a device or interface module.

When a device process executes a doio, it is removed from the ready queue and another ready process is selected for execution. An interrupt from a device causes preemption of the executing process and resumption of that device's driver (after being placed at the head of the ready queue).

2) Mutual exclusion around interface modules is no longer free. A priority queue is associated with each interface (but not device) module and contains an entry for each process waiting to enter that module. This was not needed in the original implementation because an interrupted process P1 always receives control of the processor immediately after it is released by the interrupting process P2; this is not however the case here since P2 could send a signal to another process P3 whose current priority is higher than that of P1 thereby giving P3 control of the processor. Note however that device modules do not need any mutual exclusion. This is due to the following rules:
1) when a regular process enters a device module, its priority is raised to the priority of that module [1].
2) When the processor is at priority level i, all interrupts of priority less or equal to i are masked off [1].
3) when a regular process exits a device module, its priority drops to its original value, and the process at the head of the ready queue is given control of the processor. This may result in a context switch.

The rationale of the above extension is oriented towards solving the following problems:

Process Starvation

In the original implementation process starvation can be caused in two ways: a) A process "runs away" with the processor. b) A process sends a signal to its predecessor in the ring which executes some code and drops into a wait. And this is repeated indefinitely.

In our implementation the first situation is in general not possible since the processor is not necessarily returned to the interrupted process immediately after it is released by the interrupting process. The second situation is due to the way processes are scheduled off the ring and is not possible here since we have a priority scheduling. It is however possible that a low priority process is starved if it is indefinitely overtaken by higher priority processes.

Restrictions on device processes

Wirth's implementation has a peculiar restriction on signals emitted by device processes: they cannot be received by other device processes. In previous examples (e.g. card reader to line printer stream [3]) this restriction was overcome by the insertion of a regular process between the sender and the receiver. Although this process' function is trivial and its code is very short, the solution is nevertheless non-intuitive and incurs some overhead in context switching.

If device processes were allowed to exchange signals then software administration of process descriptors would be needed. In Wirth's implementation, however, device processes are scheduled entirely by the hardware. This is a very efficient way of handling device processes. The priority mechanism described above allows device processes to exchange signals at the cost of linking and delinking their descriptors from the ready queue.

Supervisory Processes

One of the shortcomings of Modula is that supervisory processes, that is processes that schedule the execution of other processes, cannot be programmed. In fact Wirth's implementation precludes a device process from interrupting a regular process and giving the processor to another regular process. This is due to the fact that the language lacks any forceful means to switch the processor from one regular process to another. The motivation for this is to do away with mutual exclusion.

In our implementation we have sacrificed free mutual exclusion around interface modules for a better control on processor allocation. Although this does not solve completely the problem mentioned above (for a more complete solution see [6]), it enables the user to descriminate among regular processes through the use of priroties. Further, as indicated in [5], it makes the estimation of the execution time of programs more manageable.

Our priority scheme has two shortcomings:

Axiom of the wait primitive

Wirth pointed out in [5] that "... if a process would not immediately be resumed after signal receipt, no guarantee could be given for the condition Ps [associated with the signal s] still to hold when at a later time the waiting process obtains the processor ...". Our scheme does not guarantee the axioms governing signals sent by regular or device processes. In Wirth's implementation this is only a problem when the signal is emitted by a device process. Wirth gives in [5] two additional constraints that solve this problem. We found these restrictions to be too strong. In [7] Wirth mentions a simple

solution which consists of implementing wait(s) as:

repeat wait(s) until Ps

where s is the signal being sent, and Ps is its associated condition.

Larger Kernel

The implementation of mutual exclusion around interface modules has cost us some (not too significant) increase in the size of the kernel. Currently it is approximately 250 words.

Other changes and extensions are:

- A device process may specify a rank in its wait statement
- There is no lexical nesting of interface modules. The implications of this construct in the original definition were unclear.
- Records may have variant parts (to bypass strong typing).
- Structured constants can be defined.
- A CASE statement may contain an OTHERWISE clause.
- Some standard procedures for run-time debugging have been added.

4. Run-time debugging

Two standard procedures called TRACE and OUTTRACE are provided to enable the programmer to examine the values of certain variables and the status of signal queues. These procedures are part of the kernel, and can be invoked anywhere from a program. Optionally, the compiler can perform array index checking and comparison of records. Other run-time checks are:

- predecessor and successor of an enumeration value exists.
- argument of CHR is within range.
- In an array assignment, the index range of the source equals that of the destination.

5. Performance

The speed of the first phase (i.e. source to P-code translation) is roughly 180 lines per minute. The 2nd phase (i.e. code generation) is slow, (approximately half the speed of the first phase) and efforts are undertaken to improve its speed. In an experimental attempt to determine the speed of the entire compiler, we timed the compilation of a 578 line program. The translation phase took 3.09 minutes, while the code generation phase took 5.03 minutes. This corresponds to an overall speed of about 73 lines per minute. It must be pointed out that an important factor in the slowness of the compiler is that UCSD Pascal is interpreted, It runs about six times slower

than optimal.

The execution speed of PL Modula programs is quite reasonable. We estimate that the compiler produces code which, on average, is within a factor of 3 times slower than optimal. Code for variable assignment and evaluation of arithmetic expressions is very nearly optimal.

The overhead associated with the synchronization operations is as follows:

```
WAIT 153 microseconds
SEND 169 microseconds (* signal emitted to a process of >=
                         priority *)
DOIO 41 microseconds
```

6. Conclusion

Developing a Modula compiler on a personal computer has been a valuable experience for us. We learned a great deal on how to cope with memory limitations. We also found out that Wirth was very clever in the compromises he made in his implementation of the Modula kernel. It was very difficult to relax some of the restrictions he had without paying a price for them. We beleive that our extensions are worth the price we payed for them.

7. Acknowledgements

A number of people have contributed to various aspects of the PL Modula project. We wish to acknowledge the following people for their contribution to the project: Frans Heymans, Charles Hill, Dan Lorenzini, Paul Rutter, Karen Schroeder, Kees Smedema and Jerry Sullivan.

REFERENCES

1. N. Wirth, "Modula: a Language for Modular Multiprogramming" SP&E, Vol 7, 3-35 (1977)

2. N. Wirth, "Design and Implementation of Modula", SP&E, Vol 7, 67-84 (1977)

3. N. Wirth, "The use of Modula", SP&E, Vol 7, 37-65 (1977)

4. J. Holden and I.C. Wand, "Experience with the Programming Language Modula" Proceedings 1977 IFAC/IFIP Real Time Programming Workshop, Pergamon 1978

5. N. Wirth, "Towards a Discipline for Real-time programming", CACM 20,8

6. I.C. Wand, "Dynamic Resource Allocation and Supervision with the Programming Language Modula", University of York Computer Science Tech. Rep. No 15, Aug 1978

7. N. Wirth, private communication