SOFTWARE DEVELOPMENT SUPPORT SYSTEM FOR ADVANCED AVIONICS APPLICATIONS INCORPORATING A PARALLEL MACHINE ARCHITECTURE

Russell F. Vaughan and Mark S. Anastas

The Boeing Aerospace Company, P.O. Box 3999, Seattle, Washington 98124

ABSTRACT

The application of low-cost microprocessor technology to the demanding computational problems of advanced avionics challenges the ability of computer architects to develop schemes for effectively allocating processes across collateral ensembles of processing elements. There is in fact a spectrum of challenges extending into the verification and validation support for the processes so allocated.

This paper addresses the problems encountered during the development of software to be used on special-purpose parallel architectures and describes a demonstrable general purpose parallel software development support system developed at the Boeing Aerospace Company. It is proposed that with such support systems it may at last be feasible to exploit the tremendous potential of parallel processing in critical advanced avionics.

INTRODUCTION

The epithet, "special-purpose parallel," with regard to computer architectures is a description which has become synonomous with the negatives: "Difficult to understand," "impossible to schedule," and "expensive to program." Its antithesis, "general purpose" computing, has come to encompass a range of capabilities without regard to hardware. This includes the ability to support source programming in one of several conventional higher order languages, Ada constituting a synthesis for the DOD. Thus to maintain general purpose applicability claims for parallel processing architectures, it is necessary to demonstrate that programs can be written (and verified) in conventional higher level programming languages and yet run effectively on the particular parallel architecture.

The authors have defined and implemented this kind of generalized parallel software development support. It is shown in figure 1. This system was designed to support Transition Machine[1,3] developments but has more general applicability. Its purpose was to extend the range of usefulness of the Transition Machine parallel computer architecture by enabling program-development typical of general purpose sequencial computers.

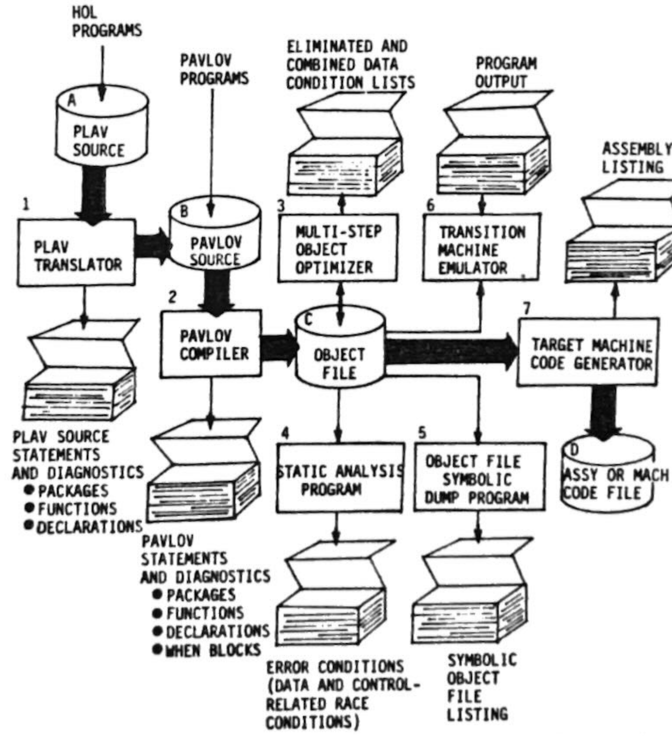


Figure 1: Parallel Software Development Support System

SYSTEM OVERVIEW

The system provides analyses pertaining to parallel program correctness in addition to its central role of translating/compiling a typical structured language, optimizing the result appropriate to a multi-tasking environment, emulating parallel machine execution and generating code for a target parallel machine. It is an assembly line along which software, ultimately to be run on the parallel machine, can be developed.

Software Development Process

The software development flow begins with a source program employing commonly used conventional language constructs. These programs are translated into an intermediate source language more suited to parallel realization of the program.

The intermediate language is characterized by a situation/response control structure that maps directly into the structure supported by Transition Machine architectures. It is a language based on a model of parallel computation[2], and is therefore applicable to parallel processing generally. A compiler generates a set of tasks and associated task control data, which together represent the source program. An optimizer reduces the level of complexity in the task contol data.

The task control data can be analyzed to identify all race conditions (both data- and control-related), compute the execution time for each task, and compute the maximum number of tasks that can be eligible at any one time.

An interpretive execution capability is provided which is modelled after the VAX 11/780 Pascal DEBUG program and provides interactive support to control the interpretive execution process.

Limited code generation exists for the Transition Machine, primarily useful in initializing task control data.

A symbolic listing can be generated at any point in the development flow.

Support System Design

The support software system shown in figure 1 is written in Pascal and is hosted on Boeing's Microprocessor Development Support Center's VAX 11/780. Every component uses the recursive descent technique described by Wirth[5], whose PLZERO compiler structure has been adapted to provide a template for each of the various component developments. The major computer program components are the following:

1. PLAV translator,
2. Pavlov compiler,
3. Multi-step optimizer,
4. Static analysis program,
5. Object file output program,
6. Parallel machine emulator, and
7. Target machine code generator.

The use of an intermediate source language and object file supports future language translation and machine retargetting requirements.

PLAV TRANSLATOR

The PLAV translator converts conventional program structures into the situation/response structure of the intermediate language, Pavlov. It performs data flow analyses to establish a partial ordering among the source statements, so that they can be activated asynchronously as tasks. The activation situation for these tasks (or "when blocks" as they are called at this stage) is specified in terms of two types of "data condition". The two types are the availabilities and updateabilities of parameters represented in the data base.

Parameters are defined as those transient realities associated with variables by assignment statements or other data type-determining operations. Multiple assignments to the same variable result in multiple parameters of the same name but different instance number, e.g. A_1, A_2, and A_3 would represent three successive parameters associated with a variable, A.

PLAV Syntax

PLAV syntax epitomizes capabilities shared by several contemporary languages. Abreviated syntax diagrams shown in figure 2 are characterized as follows:

1. data structure types: scalar, vector and matrix,
2. simple data types: integer and real,
3. file types: sequential input and output,

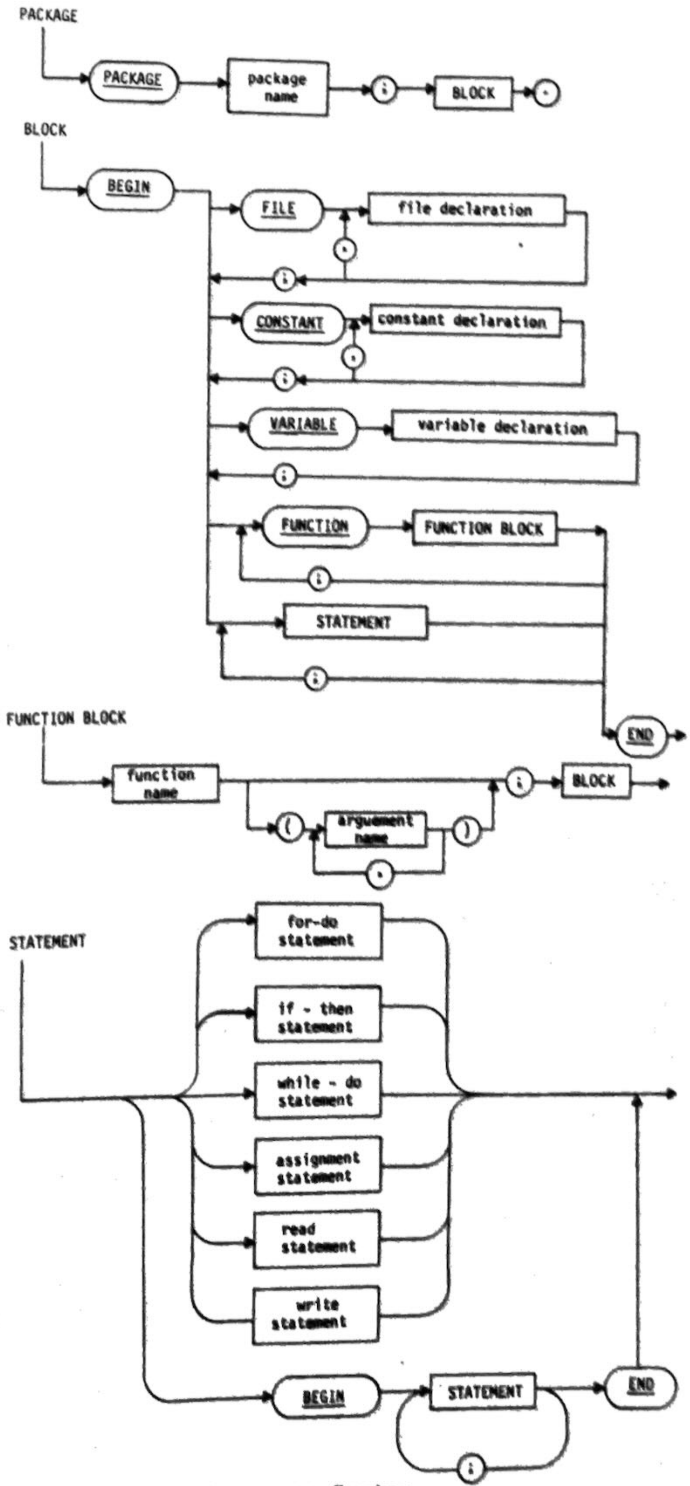


Figure 2: PLAV Language Syntax

4. functions: single-valued, multiple arguements, and
5. statements: assignment, while_do, if_then, for_do, read and write.

Assignment Statement Translation

Control relationships among statements in PLAV programs are sequential unless otherwise indicated by a control statement: If_then, while_do, or for_do. Precedence among the when blocks in the resulting Pavlov program on the other hand is determined by explicit reference to the availability and updateability of referenced or assigned parameters.

The appropriate partial ordering imposed on the when blocks associated with assignment, Read and Write statements is determined by four eligibility criteria:

1. An assigned (or written) parameter must be updateable,
2. All referenced (or read) parameters must be available,
3. The "preceding" parameter (if any) associated with an assigned variable must be available, and
4. Any parameters whose assignment references the preceding parameter must be available.

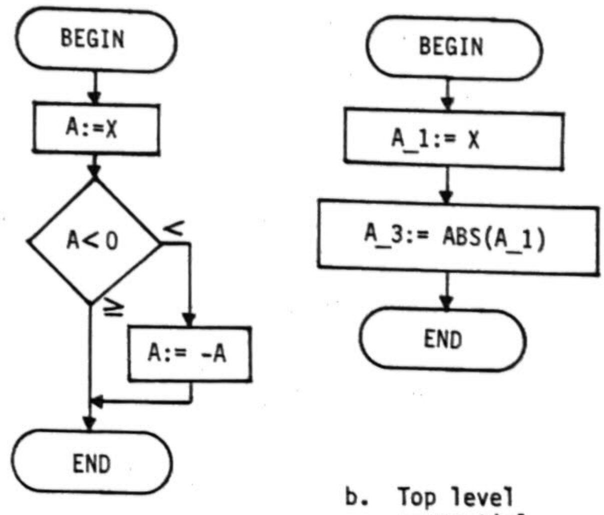
Completion of an assignment statement results in its corresponding parameter being made available and not updateable.

Logic Statement Translation

The automatic translation of logic statements is more involved, and therefore the concepts of hierarchy and the more rigorous theory of types[4] have been employed to deal with the complexity.

The elimination of GOTO statements assures that when viewed at some level of abstraction, every conventional HOL program can be visualized as a single sequence of statements beginning at the top and concluding at the bottom. The significance of this is that each statement as seen in this higher level perspective can be treated as a set of parameter assignments rather than as a set of complex logic functions. This is shown in figure 3. Each program defined at a lower level is then treated in the same way until finally a level is reached at which all statements are actual assignments.

The decision block in a conventional flow chart will be translated into an "evaluator" when block without a variable assignment statement, but with the evaluated logical expression value (A<0 in figure 3) assigned to certain data conditions, and its negation (NOT A<0) assigned to others. The complementation of A in figure 3 will translate into a separate when block at a lower level in the hierarchy. It will be a consequence of the truth of a data condition (an updateability in this case) whose value is assigned A<0.



a. Control flow

b. Top level sequential flow

Figure 3: Typical Absolute Value Computation

Even though A_3 in figure 3 is equal to either X or -X depending on the sign of X, A_3 belongs to a unique type since it has a different range of values than A_1. Whichever of the two possible relations exist between X and A_3, a new parameter assignment has in effect taken place in creating A_3. It is a value with unique properties. In this example, A_1 can be either positive or negative, and A_3 has the property of absolute values, i.e. positive.

In the original theory of types[4], it is clear that data typing is a more comprehensive subject than can be solved by merely declaring variables for once and always at the beginning of a program. A unique type is required for a variable for each instance in the program for which it takes on a unique range of allowed values. This includes not only each assignment of the variable, but also each possibility of such an assignment. In other words the fact that the variable has satisfied a test of being within a given range suffices to enjoin an associated parameter within the more restricted type just as surely as if the variable had been assigned a new value guaranteed to satisfy the restrictions.

The automatic translation of control constructs by the PLAV translator involves parameter creation in accordance with this concept of types even when no actual assignment may take place, as was shown in figure 3 for the case when A_1 is positive. Even though no assignment takes place, the variable A changes in type as indicated by its associated parameter changing from A_1 to A_3. Therefore at conclusion of either the evaluation block, or the re-assignment block, shown in the data flow diagram of figure 4, the availability of the absolute value of X is guaranteed.

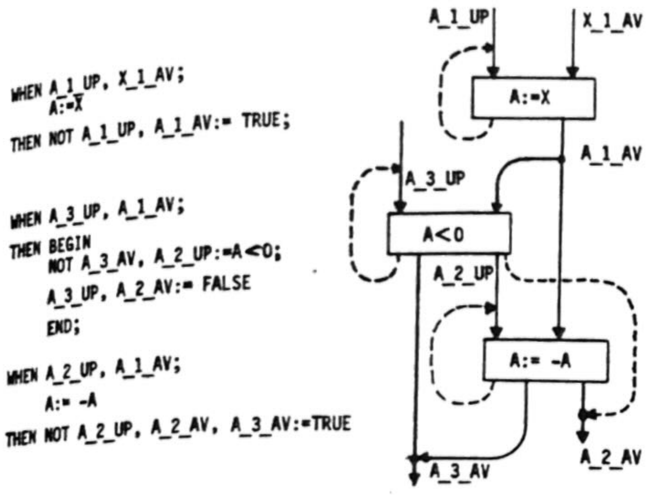


Figure 4: Translated Computation

The output of the PLAV translator is included in figure 4. This is a Pavlov program segment; the syntax will be explained further on. Data condition naming conventions should be obvious from the previous discussion.

For_do statements are converted directly to a combination of while_do and assignment statements. While_do statements are therefore the only looping construct to be translated into Pavlov. An evaluator when block results as it did for if_then statements, but this evaluator when block will have two disjunctive activation situations:

1. An initial activation occuring when all referenced parameters are available and all generated parameters are updateable, and

2. An activation occuring when all consequent parameters become available.

Figure 5 illustrates the form of a few iterative calculations with PLAV source and an optimized data flow diagram.

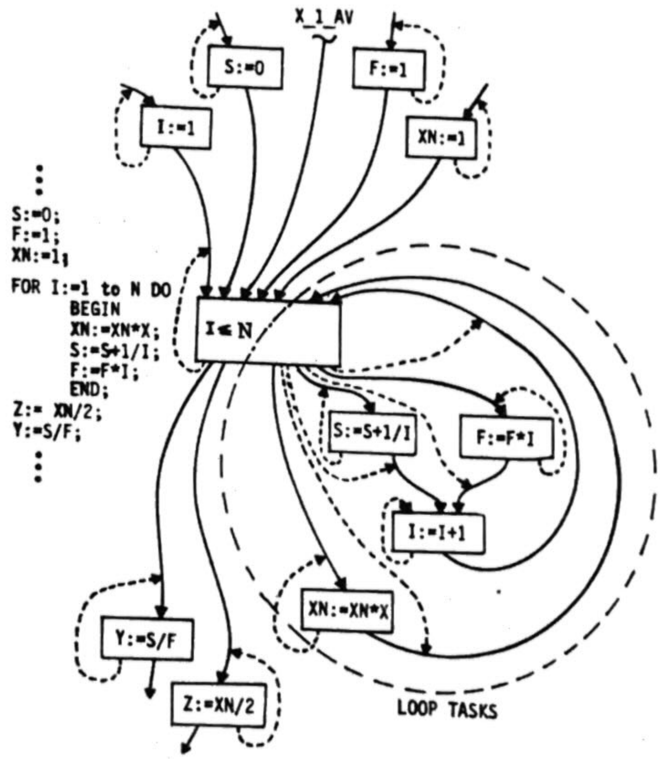


Figure 5: Translating Computation Loops

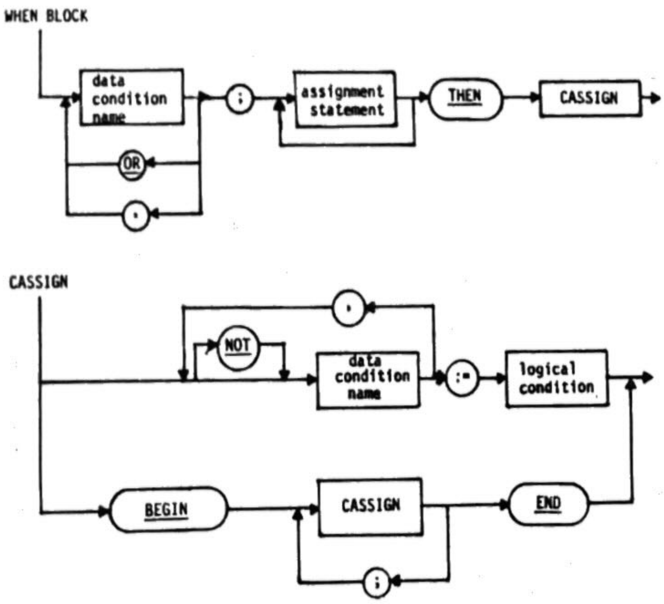


Figure 6: Pavlov Language Syntax Differences

PAVLOV COMPILER

The Pavlov language derives its name from its situation/response structure. It is applicable to rule-based systems and parallel processing generally.

The package, function and data declaration syntax of PLAV shown in figure 2 is shared by Pavlov. Boolean AVAILABILITIES and UPDATEABILITIES are added, however, and instead of STATEMENT as shown in figure 2, WHEN BLOCK syntax is implemented. See figure 6. The structure is recognizably similar to the abstract model of parallel computation called "named transition systems"[2]; the primary difference being a "then" statement.

When Block Processing

The Pavlov compiler generates tasks to be run on parallel architectures. There is a one-to-one correspondence between the when blocks of the Pavlov source program and resulting tasks. Each generated task is comprised of three parts as is the when block: The "when" statement, the variable assignment statement, and the "then" statement.

The when statement specifies a list of data conditions whose truth is required for the associated task to be eligible for execution. Data conditions must be specified in a positive sense; if the negation of a condition is required in the when statement, an opposing data condition must be defined. Disjunctive activation situations may be specified in a when statement, any one of which will activate the task; loop structures are implemented this way.

The variable assignment statement is comprised of Pascal-like assignment statements, null assignments being allowed to accommodate data condition updates without necessitating variable assignments.

The then statement specifies the modification to the status of certain data conditions as the result of executing the associated task. There are four possible dispositions for a data condition: 1. forced true, 2. forced false, 3. unaffected, and 4. forced to a value that depends on the logical relationship of variables.

Object Program Form

Task eligibility and data condition status update dispositions are incorporated into task control data. The variable assignment statement and the assignment of logical expressions to data conditions are functions for which instructions are generated for a pseudo machine. These represent the body of the task. A set of read, write and execute pointers are defined which provide a context for the task; these provide additional task control data. The form of the resulting object file is directly executable by the emulator and translatable for execution by the target machine.

MULTI-STEP OPTIMIZER

Object programs resulting from translation and subsequent compilation of conventional HOL source programs may contain excess task control information. For example, there may be overly complex requirements specifications for the tasks, data conditions declared and updated but not required by any task, or equivalent data conditions, all but one redundant. Typically also, access pointers will be duplicated.

The multi-step optimizer is concerned exclusively with task control optimization, e.g. minimizing the number of (and references to) data conditions. No attempt has been made to optimize the pseudo code comprising the tasks, although access pointer storage has been minimized. The optimization takes place in a number of separate phases or steps, each performing a distinct function. The phases can be applied in any order (and to varying depths). The order of their application affects only the efficiency of the optimization, not the structure of the result. However, the specified degree of optimization affects the quality of the product.

Unnecessary Data Conditions

A data condition is unnecessary if it is not required by any task. In such cases the data condition affects the eligibility of no task and can therefore be deleted. In figure 4, the data condition A_2_AV is an unnecessary one. A_2 is the parameter corresponding to minus X. But since A will never be accessed in this capacity, A_2_AV is unnecessary. Such data conditions, and all references to them, are eliminated from task control data.

Equivalent Data Conditions

Two data conditions, always updated in the same sense, represent the same information. In this case the optimizer deletes one of the data conditions, and substitutes references to the other. Where duplications are induced, subsequent reorganizations eliminate them.

Combined Data Conditions

An updateability data condition is included for every when block. Its purpose is to turn the task off once it has been activated, precluding inappropriate contemporaneous activation by another processor. Where an availability data condition is also required exclusively by the task, it could be used to provide this turn off function. Negating its availability would negate the eligibility of the task without affecting the eligibility of others. In such cases the updateability data condition is eliminated and the availability data condition's role is extended to include that of the updateability.

Simplifying Requirements

The when statements specify situations under which task execution is appropriate in terms of referenced parameters being available, etc.. Since certain parameters may involve certain others as prerequisites to their assignment, activation situations are typically over-stated. Therefore some data conditions in the requirements list can be deleted without affecting the partial-ordering of tasks.

In the first of two classes of overstated requirements shown in figure 7, the requirement for the data condition C1 by task T3 can be eliminated because the requirement for C2 insures T1, T2, and T3 will execute sequentially. Similarly, the

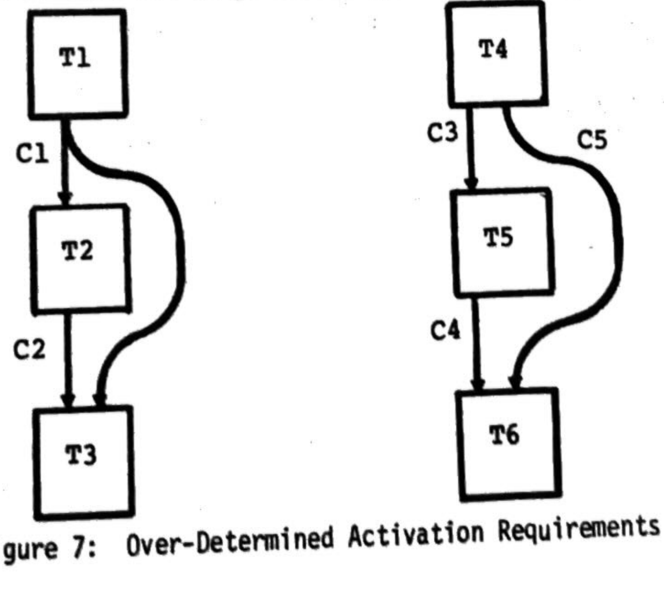


Figure 7: Over-Determined Activation Requirements

requirement for C5 can be eliminated since the requirement for C4 guarantees the correct ordering of T4, T5, and T6 without necessitating that T6 require C5. At least one other data condition that supports the "turn off" ("not updateable") function must be found, however, before such data conditions can be eliminated.

STATIC PROGRAM ANALYZER

Static analysis of task control data generated by the Pavlov compiler practically provides an x-ray of the source program's structure. It reveals race conditions and maximum and minimum task concurrency.

List of Significant States

The key to this analysis is a list of significant system states that are reachable from a given initial state. The number of reachable states for a system may be very large, up to 2^n for n data conditions. Fortunately, only a minor subset of these states are actually reachable, and of these, very few are pertinent to static analysis procedures.

From any state a one-step reachable state is one which can be realized by completing one of the eligible tasks. The number of one-step reachable states from a given state is dependent on the numbers of eligible tasks and conditionally assigned data conditions.

"Subset" states are defined to minimize processing. State A being a subset of state B implies A OR B is equivalent to B. Since, therefore, every data condition that is true in state A is also true in state B, all tasks that are eligible in state A will be eligible in state B, and race conditions in A will exist also in B.

A list of significant states is obtained by examining all one-step reachable states starting at the initial state declared in the Pavlov program. Those states which are not subsets of other reachable states are retained. Sets of concurrently eligible tasks are generated during this process, as are the maximum and minimum numbers of eligible tasks.

Race Conditions

A system of tasks contains race conditions if the order of executing concurrently eligible tasks in the system affects any required computation. There are two types of race conditions identified by the static analysis: Data-related and control-related.

If a task assigns a contemporaneously shared variable, a data-related race condition exists. If any data condition required by one of a set of concurrent tasks is reset by any other of these tasks, a control-related race condition exits.

PARALLEL MACHINE EMULATOR

An interpretive emulation of a Transition Machine provides an interactive support tool for debugging programs developed on this system. It generates an executable image stored internally to the interpreter, which emulates the behaviour of virtual stack machine processors and a multi-tasking executive mechanism. This mechanism is the System Controller[1] in Transition Machine architectures.

Interactive Commands

The operation of the emulator is controlled by operator commands entered interactively from the user terminal. This interface is modelled after the VAX Pascal debugger provided by DEC. The following functions are supported:

1. Examine the contents of variables, data conditions, registers, etc.,
2. Deposit values into variables, data conditions, registers, etc.,
3. Watch any variable, data condition, register, etc., suspending execution when modification is detected.
4. Set breakpoint at task entry,
5. List currently active tasks,
6. List currently eligible tasks,
7. Run the program,
8. Single-step the program one task at a time, and
9. Trace the program's execution.

Virtual Machine Definition

The virtual machine being emulated is comprised of three primary components: The System Controller, the processor, and the memory.

The main memory in the virtual machine is assumed accessible to both the System Controller and the individual processors. It contains the following information:

1. A status vector of boolean indicators used to maintain the status of all data conditions in the system.
2. A task control block array containing the lists of data conditions required to activate each task; the access pointers indicating read, write and execute authorizations; and update dispositions for data conditions that result from each task.
3. The pseudo machine instructions associated with the body of each task, and
4. Data storage for all declared constants and variables.

A set of virtual processors is defined to interface with the System Controller to obtain task assignments and interpretively execute the pseudo instructions generated by the Pavlov compiler. These virtual processors have a simple stack architecture, with internal registers for interfacing to the System Controller, performing arithmetic operations, and for stack storage.

Each pseudo instruction contains two fields, an operation and an operand which indicates a displacement into the access pointer list. Since there are no transfers of control or direct memory access instructions, an application task is completely contained by its access pointers set up by the System Controller.

Two registers comprise the System Controller's interface with each processor: One specifies an index into the task control data , the other transfers data condition values when they are determined by the task as a logical expression assignment.

The virtual System Controller operates on the task contol data and data condition status vector. In effect it is a multiprocessor executive handling event relatonships exclusively.

EXAMPLE PROGRAM DEVELOPMENTS

The first step in the program development process is the translation of the source program by the PLAV translator. A PLAV listing results, which if there were errors present, would contain diagnostic messages in addition to the source program. An example is shown in figure 8. Notice that line numbers are included on the listing.

```
10 PACKAGE          PRIMENUMBER;
20
30 '      THIS IS AN EXAMPLE PROGRAM WHICH COMPUTES THE GREATEST
40 PRIME NUMBER LESS THAN OR EQUAL TO N.  THIS NUMBER IS STORED
50 IN MAXPRIME ON COMPLETION OF THE PROGRAM.                    '
60
70
80 BEGIN  '         PACKAGE          '
90
100 CONSTANT         TRUEVAL=1,FALSEVAL=0;
110
120 VARIABLE         N,I,J,PRIMEFOUND,MAXPRIME;
130
140        BEGIN  '        STATEMENT        '
150        N:=25;
160        I:=N;
170        PRIMEFOUND:=FALSEVAL;
180        WHILE PRIMEFOUND=FALSEVAL DO
190               BEGIN
200               PRIMEFOUND:=TRUEVAL;
210               FOR J:=2 TO I/2 DO
220                 IF (I/J)*J = I THEN PRIMEFOUND := FALSEVAL;
230               IF PRIMEFOUND=FALSEVAL THEN I:=I-1
240               END;
250        MAXPRIME:=I
260        END    '        STATEMENT        '
270 END.   '      PACKAGE         '
```

Figure 8: Example PLAV Program

A Pavlov program is also output by the translator if no errors are detected in the source. Each when block in the Pavlov program has a label corresponding to the line number on the PLAV listing of the statement which is implemented by the when block. In the example program, the if_then statement on line 230 translates into two when blocks labeled 230 and 231.

Figure 9 provides a data flow diagram corresponding to the Pavlov program generated by the PLAV translator. Each node is associated with a when block, each interconnecting line a data condition. Incoming lines indicate data conditions required by the when block, outgoing lines data

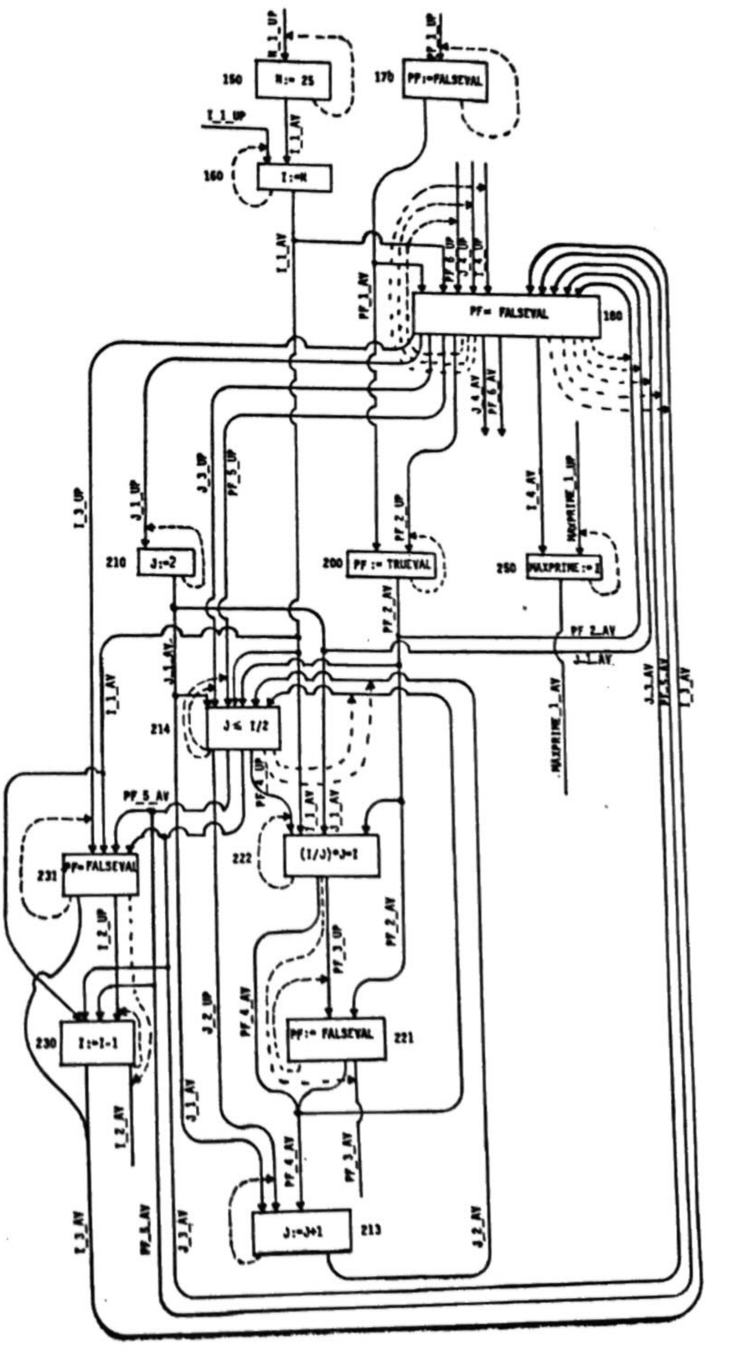


Figure 9: Data Flow Diagram of Example Pavlov Program

conditions updated as the result of the when block. Solid lines represent data conditions that are set, dotted lines those which are reset.

Following compilation, the program can be optimized. As shown in figure 10, the result is a significant reduction in the complexity of the associated data flow diagram. Optimization reduced the number of data conditions by 12, and when block requirements lists by 24 in this example. In general it has been observed that when blocks require an average of between one and two data conditions and that the entire system of tasks will contain about 1.5 times as many data conditions as there are when blocks. These numbers apply to

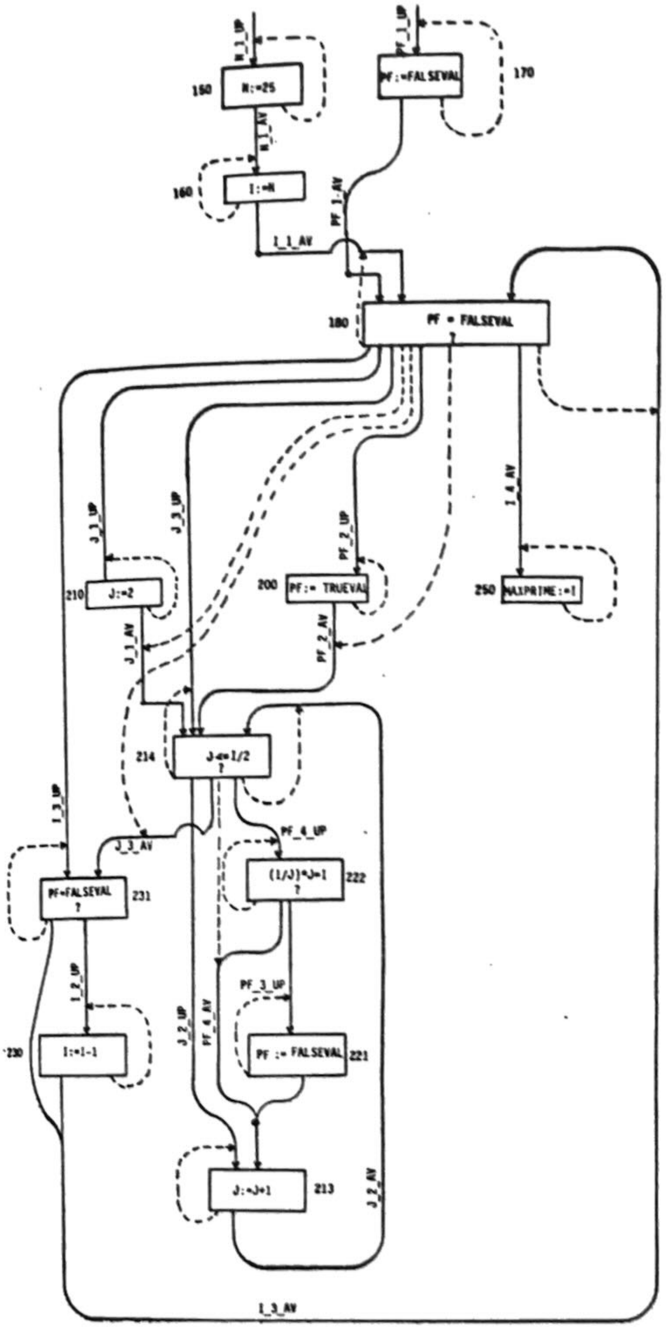


Figure 10: Data Flow Diagram of Optimized Example Program

optimized programs, and seem to be relatively insensitive to task concurrency.

Programs can then be run on the emulator to validate their execution prior to generating code for the target machine.

CONCLUSIONS

The parallel software development system is a precursor rather than a finished product. It is felt however that many of the most difficult issues have been addressed by this development, and the approach is flexible to modifications in either the source language syntax or target machine design.

It is apparent that advanced avionics applications will have much to gain by exploiting the benefits of parallel processing. With software development support as envisioned here, many of the obstacles will have been removed.

BIBLIOGRAPHY

[1] Anastas, M. S. and Vaughan, R. F., "A Prototype Parallel Computer Architecture for Advanced Avionics Applications." Proc. of NAECON '82. May 1982

[2] Keller, R. M., "Formal Verification of Parallel Programs." Comm. ACM 19,7 July 1976

[3] Vaughan, R. F. and Anastas, M. S., "Microprocessor Based Transition Machines." Proc. of COMPCON FALL '79. Sept. 1979

[4] Whitehead, A.N. and Russell, B., "The Theory of Logical Types." Principia Mathematica, Cambridge, London 1967

[5] Wirth, N., Algorithms + Data = Programs, Prentice Hall, Englewood Cliffs, New Jersey 1976