

A Vector C and Fortran Compiler for the FPS T-Series: Experiences with compiling to occam I

D. E. STEVENSON

Department of Computer Science, Clemson University, Clemson, SC 29634, U.S.A.

L. K. AMMONS

Department of Computer Science, University of Tennessee, Knoxville, TN 37920, U.S.A.

W. G. CROSMUN

Department of Economics, University of North Carolina, Chapel Hill, NC 27514, U.S.A.

A. JACKSON

Data General Corporation, Research Triangle, NC 27709, U.S.A.

AND

G. L. RAJ

Sun Microsystems, Mountain View, CA 94043, U.S.A.

SUMMARY

We describe our implementation of C and Fortran preprocessors for the FPS T-series hypercube. The target of these preprocessors is the occam I language.

We provide a brief overview of the INMOS transputer and the Weitek vector processing unit (VPU). These two units comprise one node of the T-series. Some depth of understanding of the VPU is required to fully appreciate the problems encountered in generating vector code. These complexities were not fully appreciated at the outset.

The occam I language is briefly described. We focus on only those aspects of occam I which differ radically from C. The transformations used to preprocess C into occam I are discussed in detail. The special problems with the VPU both in terms of its (non)interface with occam I and in dealing with numerical programs is discussed separately.

A lengthy discussion on the special techniques required for compilation is provided. C and Fortran are simply incompatible with the occam I model. We provide a catalogue of problems encountered. We emphasize that these problems are not so much with occam I but with preprocessing to occam I. We feel the CSP and occam I models are quite good for distributed processing.

The ultimate message from this work should be seen in a larger context.

Several languages—such as Ada and Modula-2—are being touted as the standards for the 1990s. These languages severely restrict parallel programming style; this may make saving dusty decks by preprocessing an impossibility.

KEY WORDS Parallel programming C CSP Compiler design Distributed systems

0038-0644/92/050371-20\$10.00
© 1992 by John Wiley & Sons, Ltd.

Received 13 December 1989
Revised 16 December 1991

INTRODUCTION

In September 1986, Clemson University purchased a Floating Point Systems (FPS) T-series computer. The T is a hypercube of vector processors: that is, each node is a vector processor. When the T arrived, it had only the occam I language available. In order to make the T usable for numerical work, at least a Fortran compiler was required; certainly for system work, a C compiler was called for. While FPS and Cornell University had already started down this path, local researchers felt that a 'vanilla' Fortran or C was not going to solve the programming problems which were anticipated; in fact, it might actually create more problems than they would solve because of the catastrophic changes in the virtual machine presented by the T would make using a standard sequential language difficult. Additionally, the numerical analysts were interested in a vectorized version of Fortran, but one which made the algorithms clear. This meant that we needed to extend the standard by including linear algebra primitives as primitive operators.

With that in mind, we decided to develop our own distributed memory C and Fortran which would incorporate some of the CSP principles developed by Hoare¹ and CCS of Milner.² For various reasons, we did not choose to implement either C++ or Fortran 8x. Clearly we could bootstrap C++ if we could get C implemented. The Fortran 8x issue was much more complicated: the computer science researchers thought 8x would be the way to go; the mathematical sciences researchers felt it would take too long to implement. The telling argument, though, for both sides was the realization that 8x is vector oriented and not distributed memory oriented. We chose something much closer to Fortran 77, feeling that it would be more useful (read 'marketable').

In order to take advantage of the graduate students' interest, the compiler class taught by one of us (Stevenson) was encouraged to 'volunteer' to write a preprocessor from C to occam I.* The approach was to use the portable C compiler grammar³ and develop a simplified VAX Fortran grammar for *yacc*.⁴ For both compilers, we wanted to address two major requirements:

1. We wanted to have a distributed memory language rather than a von Neumann language forced to think it was distributed by using subroutine calls.
2. We would modify the language to express vector processing directly. The T has a full repertoire of vector operations available to it—the numerical analysts were anxious that these be fully used.

THE FPS T-SERIES

The Floating Point Systems T-series computer is a hypercube of processors. Each nodal computer is both a standard processor and a vector processor. These nodal processors are networked together as four dimensional hypercubes. In this section, we describe the hypercube topology and the vector processor node which consists of an INMOS transputer T414 and a Weitek Vector Processing Unit, VPU (see [Figure 1](#)).

* Occam II would have been the language of choice for this work. Unfortunately, it was not an option due to Floating Point's ongoing development in occam I.

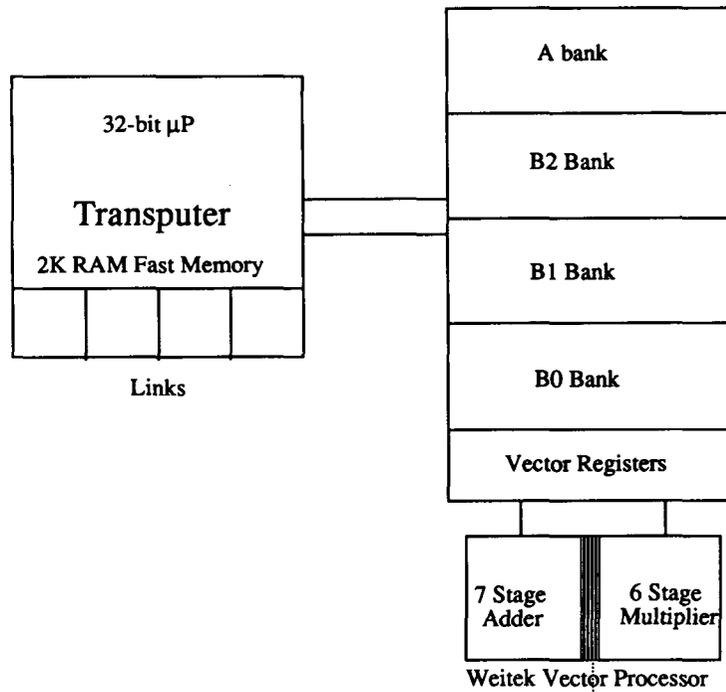


Figure 1. Vector processor node for Floating Point Systems' T-series

The INMOS T414 transputer

The INMOS T414 transputer is considered a RISC machine in many respects. For example, the system has only three work registers and a limited repertoire of bit, byte, integer instructions. The T414 has no floating point unit but the newer version (the T800) does. The transputer paradigm allows for four hardware channels which can be attached to the outside world. In the T-series hypercube, we have the nodal computers connected through these hardware links. The major impediment to communicating with the outside world is the lack of interrupt capabilities in the hardware. While experience with transputers shows this lack is not a detriment for one transputer operating alone, it is critical for communicating with transputers which do not have fixed communications topology. Times to reset the links of the T are very long compared to communication times: one has to wait until the other end recognizes the request to relink.

INMOS has made an effort to preserve occam I as the sole native language* of the transputer and to develop a central system based on occam I. In all fairness, this seems to have been a marketing ploy rather than the feelings of the technical staff. We discuss occam I and the *harness* concept. Be that as it may, the recalcitrant marketing view greatly restricts the use of the transputer—especially when something unplanned for is added to the hardware. INMOS has since modified this stance, but too late to save the T-series.

* Occam is not an assembler level language. It might be favorably compared with C with respect to level and C's relationship with the PDP 11/70 hardware. Another possible comparison is with Modula and Lillith.

The vector processor

The unplanned-for addition is the Vector Processing Unit or VPU. The essential architectural features of the VPU are shown in Figure 1. The operational units are a pipelined adder and multiplier. Memory is the same as the transputer's; however, the memory comprises four partitions: A, B0, B1, and B2. Collectively, the B0, B1 and B2 partitions are called the B bank. This is a major headache in compiling and in programming. The rules for routing data between the adder and the multiplier are quite complicated. However, the unit has quite an extensive instruction repertoire. For those not familiar with the VPU we include a brief description of the types of operations.

VPU instruction repertoire

Vector operations include the obvious, such as add, subtract, and multiply. [Recall division is not defined.] The VPU uses multiple *registers* to perform operations; consequently, there are load and store instructions. Because of the dimensionality of vectors, there are also several other type operations.

1. Reduction add and reduction multiply. These functions produce a scalar by adding or multiplying all the elements together.
2. Fill. Vector fill instructions propagate a given scalar throughout the vector.
3. Shift and rotate. A vector shift, in general, is a truncating operation wherein the vacated positions are filled. Shifts fill with indeterminate values. A rotate, on the other hand, is a circular shift.
4. Minima and Maxima. These operations find the *index* of the minimum (maximum) and return it through a special return mechanism.
5. There are several exotic operations which we do not discuss.

From the code generation standpoint, the VPU is very complicated. Only certain combinations of registers, memory, and the operational units is allowed. We explain some of the rules below to give the reader a feeling for this complexity. FPS had already developed a system of *parameter block routines* for low-level coding (see Figure 2). Nevertheless, one must fill in all the fields. An example is provided later which illustrates the use of the parameter blocks.

The transputer and the VPU share memory, but the VPU has a very rigid view of alignment: all vectors must start on 1K byte boundaries. One can, however, deal with 'quarter vectors' through 'taps'. We did not allow vectors beginning on anything but 1K boundaries. Vectors can, however, be of any length. Vector lengths are always rounded up to the nearest 1K boundary. This also minimizes overhead of keeping track of tap boundaries and performing various shifts and moves to temporary locations. This obviously increases memory requirements, but this fragmentation should not be substantial unless numerous short vectors are used.

Restrictions of vector boards

Effectively, the VPU is set up to do those operations which one finds in numerical linear algebra applications. The so-called BLAS of the LINPACK system are quite natural. Each operation assumes there can be two vectors and a scalar. There are interesting restrictions on memory bank placement of data if two vector operands

Field	Definition
VP.opcode	Operation Code which contains subfields for adder and multiplier functions plus precision and length specifications
VP.sload	Controls loading of scalar value if called for by VP.opcode
VP.srcmsk	Mask defining locations of source operands; e.g., one of the bank registers
VP.dstmsk	Mask designating where results are to go
VP.srcV1	Address of first vector operand
VP.srcV2	Address of second vector operand

Figure 2.

are used. Code generation is complicated: only certain paths between the adder and multiplier can be used. To indicate the complexity of coding the boards, here are some of the rules:

1. If the operation has two operands, one of which is a scalar, then the vector may come from either the A bank or the B bank.
2. If the operation has two vector (and no scalar) operands, then one must be from the A bank and the other from the B bank.
3. For the two vectors and one scalar case, the above restrictions plus the following restrictions apply:
 - (a) If the scalar is to be added via the adder—as in $(\text{vector} + \text{vector}) + \text{scalar}$ —the vector sum must either be the multiplier or the A bank.
 - (b) If the scalar is run through the multiplier— $(\text{vector} + \text{vector}) \times \text{scalar}$ —the product source must be either the adder or the B banks.

These restrictions necessitate a large number of temporary vectors. We chose to make copies as needed, and only as much as was needed at a time. This slows the vector processing somewhat, but greatly reduces the amount of temporary space needed if the vectors are quite large.

OCCAM I

Some features of occam I

We now turn our attention to the native programming language of the Transputer, occam 1.⁵ We present a very rudimentary introduction to occam I and hasten to add that we are not able to do justice to the language. Our purpose is to present enough information to allow the reader to understand the semantics of the language and, in particular, contrast occam I to C and Fortran. Very good texts are available which

do justice to the power of occam. ^{6,7} While occam I has been maligned on occasion, ⁸ there is much to recommend it in a distributed environment. Our principal problems with occam I is that it is *not* an assembler but a higher level language with a limited vocabulary, more like Bliss. ⁹ For that reason occam I is very difficult to preprocess into. *

It is also fair to point out that one cannot separate occam I the language from occam I-TDS, the development system. TDS—the Transputer Development System—is a software tool which serves as text editor, compiler, and linker. It is hard to separate ones comments about occam I from TDS. The concept of development environments such as TDS is very good, in our opinion. But for preprocessing, TDS presents a real obstacle. INMOS has decoupled TDS from the newer occam II.

Another concept of occam I should be mentioned, the so-called *harness* concept. In the occam I documentation, ⁵ a view is presented that occam I not be considered as the only language of the system, rather that it be the language in which parallel control be expressed. This view leaves open the possibility of having all languages available to the programmer. One would simply use occam I to control parallelism for such languages as C or Fortran. *Prima facie*, this seems to be a great idea. (Unfortunately, marketing did not seem to have read this section of the manual.) As it arrived on site, the T-series had a harness for both the Transputer and the VPU. This greatly simplified our task of preprocessing.

Primitive data types

Occam I has the usual low level primitive data types: bit strings (as a function of word size), bytes, integers, and floating point. (Our T-series has the older T414 chips and does not have on-chip floating point.) Of significance, though, is the fact that occam I does not set a precedence on the operators, so liberal use of parentheses is demanded.

Expressions and guards

For languages like occam I, one should think of ‘expressions’ in two lights. The first, and more conventional, is the evaluation of something which returns a value. The second interpretation is that of a guard. A *guard* is an expression which must be satisfied before execution is allowed to proceed. Occam I makes use of both. The guard concept is described below in relation to the ALT command.

Channels

The major benefit to using occam I is that message passing is handled by primitive occam I constructs supplemented by guards. (See below, under ‘Primitive processes—input process and output process’.) The CHAN nel is the dominating theme of occam I. There are two operators associated with CHAN nels:

```
'?' (read "input ")
'!' (read "output ")
```

* We attempt to catalogue our experiences later in the paper,

These operators can be thought of as assignment instructions which must occur simultaneously. Such *synchronous message passing* strategies have much to recommend themselves in a distributed environment.¹⁰ Occam I channels are implemented in one of two ways:

1. One class to communicate on one of the four hardware links.
2. One class to communicate between PROCesses.

The classes are differentiated by the address of the buffer control information. The transputer—as distinct from occam I—has several instructions to aid in the synchronization and message transfer.

Primitive processes

The principal organizational concept in occam I is the *process*. These processes are inherently parallel. There are five primitive processes:

1. Assignment process. The usual imperative language statement is considered an independent process.
2. Input process. The input process is one-half synchronization mechanism. The input command '?' is issued against an input channel. If the data is not already available, the requesting process waits. We emphasize that there are no testing commands; the wait, if it is to occur, is unavoidable.
3. Output process. The output process is the other half of the synchronization mechanism. If one outputs ('!') to an output channel, the data will be transferred to the 'input' ('?') process once the input command is issued. Otherwise, the output process waits.
4. Timer/wait process. The wait process continues processing once the 'alarm time' has passed.
5. Skip process. The skip is a process which can always execute and always terminates. Skips allow for an orderly termination of many constructs (see ALT below).

The input and output processes are familiar to operating system designers as 'blocking reads and writes'.¹⁰

Compound statements

There are two structures for combining individual statements into compound statements: the SEQ and the PAR blocks. In the SEQ blocks, the statements are executed in the 'usual' order as one would expect in C, for example. In the PAR block, the statements are executed in parallel and hence in an indeterminate order. The SEQ and PAR statements are also used to control data allocation in a manner reminiscent of Algol.

Syntactically, occam I does not use a closing 'end' as one might expect; rather, the scope of a block is indicated by the indentation. While this might be considered a large burden on the programmer, TDS shoulders the load. Unfortunately, reading an occam I program into TDS requires that the indentation be correct. These comments relate to the preceding comment concerning the tight relationship between occam I and TDS.

Conditional processes

There are two processes which can be thought of as equating to the if-then-else in the usual imperative languages. The IF construct takes a sequence of condition-action pairs with semantics similar to the `cond` statement in Lisp. The ALT statement uses guard-action pairs. The semantics of the ALT statement is that each condition is a guard. At any given time, the ALT 'blocks' the further execution of the actions until at least one of its guards is 'ready'. All this, of course, is the Dijkstra *guard* construct.¹¹

Repetitive processes and replicators

In occam I parlance, the term *replicator* refers to statements which can replicate a component process while the term *repetitive process* is reserved for the WHILE construct. The SEQ, PAR, ALT and IF statements may all act as replicators. The replication clause has a simple incremental counter mechanism, `<var> = <lowconstant>` for `<highconstant>`. In the version of occam I we used, the `<lowconstant>s` and `<highconstant>s` were required to be integer constants. For example, the Fortran DO statement `DO limit I = 1,20` would most naturally be translated `SEQ i = [1 FOR 20]`. Note the `PAR i = [1 FOR 20]` is probably not equivalent!

Procedural statement

Constructs may be formed into Procedures. The PROC is the only procedural type—there are no functions. Formal parameters may be declared and the declaration must specify whether or not a parameter is passed by value (VALUE parameters), by reference (as VAR parameters), or a CHAN nel. Of crucial importance is the restriction that one cannot pass a vector starting at other than the first location; e.g. the statement `foo(vec[3])` is not allowed for a procedure `PROC foo(value x[])` or `PROC foo(var X[])`.

Scheduling

A two-level round-robin scheduling strategy is used by the system. The transputer (at the 'machine' level) supports the generation and control of the parallel processes. The two levels should be thought of as 'fast' and 'other'. The use of the prefix PRI on the ALT or the PAR cause the first (textually) component process to take precedence in ties.

Summary and example

The syntax and semantics for occam I are quite simple and straightforward. While some of the constructs have quite different semantics from either Fortran or C, it must be remembered that the occam virtual machine is quite different as well. A simple illustration of the types of differences one might see in an occam I program would revolve around the use of (1) PARs and SEQs and (2) CHAN nels and the two message operators, '?' and '!'. The program below is a parallel program which reads an input value from channel c1, puts the value on a channel comms, takes the value off of comms and outputs to channel c2 (see [Reference 5](#), p. 3.4.2)

```

CHAN comms:
PAR
  WHILE TRUE
  VAR x:
  SEQ
    c! ? x
    comms ! x
  WHILE TRUE
  VAR x:
  SEQ
    comms ? x
    c2 ! x

```

FORTRAN AND C SEMANTICS AS TRANSLATED TO OCCAM I

One might think that the translation of C and Fortran to occam I would be comparatively straightforward undertaking. After all, occam I has a fairly complete repertoire of constructs and data types. Such is not to be. In its favor, occam I is a language that was designed to assist *humans* in programming in parallel. At one time, it was reputed to be the most widely used language in writing programs for multiprocessor systems. Therefore, there is a certain amount of societal experience and knowledge. However, this knowledge does not transfer well to a preprocessor. In a later section, we discuss specific criticisms of *occam I as a target for preprocessing* only.

There were some simple changes to C and Fortran to bring them in line with the parallel environment. For C, these are

1. The primitive type CHAN to support channels, along with ‘??’ for input and ‘! !’ for output. We had to go to ‘! !’ because ‘!’ was already taken—such is the way of extensions. These functions were added to the grammar as assignment operations.
2. The pair of symbols ‘{’ and ‘|}’ to indicate blocks of code to be run in parallel; i.e. the PAR. The ‘{’ and ‘}’ are retained for sequential blocks—the SEQ.
3. The primitive type vector is added to differentiate from ‘array’ or ‘table’ concepts.
4. Extended operator symbols for enhanced operations, such as compression operators.

The same changes were made to Fortran, except parallel and sequential replace the equivalent symbols in 2, above, and end for the right braces. In addition, certain primitive functions needed to deal with the transputer’s links were included at compile time. This removed the need for the C and Fortran compilers having special initialization and new include libraries. The harness concept discussed above made the inclusion of environmental information easy.

Benefits of preprocessing to occam I

There are obvious benefits to compiling to a higher level language. For the most part, scalar integer operations coded in a rigid manner in C or Fortran translate

directly with little change. Some annoying problems did occur; for example, there is no precedence among operators, so everything was parenthesized. However, one need not get into temporary results and the concomitant temporary variable allocation in most cases.

Since occam I already had the mechanism in place for handling scoping and activations, we did not do so directly. This ultimately was a mistake; however, the lack of memory management facilities made this decision for us. This is discussed below. Of course, we did need to keep up with blocks and the visibility of variables within the block, as well as the indentation occam I required, but we did not keep track of memory requirements nor did we allocate space for scalars. It was necessary to allocate vector space according to the rules outlined above.

The occam I compiler and T414 do not handle real numbers. This leads to a very bizarre problem of allocating and manipulating floating point numbers. The Weitek VPU uses the IEEE standard for floating point. We had to translate the floating point numbers to hexadecimal. For double precision, we needed two such words. To add to the problem of debugging, the double precision numbers were stored in a two word vector in *reverse* order—a protocol forced on us. To get the pair of integers to IEEE format, we called various functions provided by FPS.

How C statements were implemented

For the standard assignment statements, without functions and using the occam I primitives, the semantics are quite clear. The C expressions translate directly, needing only to produce the parenthesized occam statements. For expressions with function calls, the function must be evaluated prior to the computation of the expression. This is no great inconvenience, although one would hope that preprocessing from one higher level language to another would not be so painful.

In order to support the C for statement, we use the occam I WHILE. Here again, things are much more difficult than one would hope (expect). One needs three separate operand stacks in addition to the normal processing (statement) stack. The initial statement is copied into a *pre*-stack and the increment statement is copied into the *post*-stack. In general, the *conditional* statement is the condition on the WHILE statement. The normal mode for generating code would be to produce the initialization from the *pre*-stack, condition, statement and increment from the *post*-stack. But, again, this is not simple: break and continue require a stand-alone variable, initially set to TRUE. To fully appreciate the job required for breaks, please see the discussion below on goto s.

For the if-then-else and the switch, the occam I IF does nicely. We did not try to use the prioritized versions. As noted before, the break and continue statements cause the translation to be quite unclean. We did rearrange the C grammar to disallow cases and defaults outside of the context of the switch. To see how breaks function in this context, note that the occam I IF statement—unlike the C if statement—supports multiple cases that can be evaluated at run time. Also, unlike the C switch statement, only the expressions in an occam I IF statement that are true are executed, and *all such* expressions that are true are executed. In order to support the C uses of break and to restrict execution to the first true break we generated two boolean variables, Occam I Break, and Occam I Default. Then if is translated into

IF
(case expression) and (not Occam I Break) . . .

The C break statement generates the occam I statement

Occam I Break: = TRUE

Unlike the C break statement, the occam I BREAK statement will not halt execution. This was deemed a small price to pay for simplicity. One side-effect of this is that, while C does not support run-time evaluation of the individual case relations statement, we do. This makes for somewhat more usable switch statements.

Structures and pointers were implemented. FPS had implemented a memory management allocation/deallocation mechanism. We did not try to implement recursion.

VECTOR SEMANTICS

Changes to Fortran and C to support vector processing

The main goal of the work described here was to develop a Fortran and C compiler which would take advantage of the VPU without having to call external subroutines. That is, we wanted to add a vector processing capability to the languages and still control the code. The ANSI X3J3 subcommittee—the so-called Fortran 8x*—definition was available to us. At first we considered implementing the standard; after some discussions with both the mathematicians and computer scientists, this goal was dropped. The C compiler was a more ‘vanilla’ version; however, we did provide for simple vector expressions.

Our basic approach was to make two simple changes to Fortran syntax. The first was to adopt the indexed triple notation found in some Fortrans, most notably VAX Fortran. In the index triple, we allow for coordinates with the form

lowvalue : highvalue [: increment].

This construct expands into a DO loop with the appropriate parameters set. Where practical, the vector processor is used.

The second change is more imperative. In this form, we remove any *expression* reference from the co-ordinate. This is replaced by a simple *dash* (‘-’) read ‘missing’. This gives the compiler free rein to convert the expression for use on the VPU. For example, the VPU has the capacity of 128 double precision floating point words in each of its registers. Thus,

$$x(-) = A(-) * B(-) + s$$

is processed at the maximum rate. An initial constraint on the syntax was that there could be any number of missing co-ordinates, but all must start with the first and no gaps. That is,

* During development, the 8x standard has not been adopted. Even if it were, it would not support distributed processing.

$$x(-,-) = A(-,-) * B(-,-)$$

and

$$x(-,i) = A(-,i) * B(-,j)$$

are both legal but

$$x(i,-) = A(i,-) * B(j,-)$$

is not.

The temporaries problem

The section on restrictions to coding the VPU does not begin to convey the complexity of coding the VPU. This is demonstrated in [Figure 2](#). There are two more problems, mostly concerning memory. When we first received the T, there was effectively no operating system and precious little documentation on whatever was there. A critical deficiency was the lack of memory management. This was overcome, with the obvious ploy: we implemented it. The second memory problem concerns the placement of data: all vector operations taking two operands require one operand—sometimes a specified one—to be in the A-bank. The A-bank is one-quarter of memory: 256K. This makes the A-bank a very critical resource. The problem is complicated by the necessity to create temporaries—also in that A-bank.

Our approach to minimizing the A-bank space is, to break the computation down into slices. While the term ‘strides’ commonly refers to the increment in the triple notation described above, we use the term ‘slices’ to indicate a 1024 byte computation. For example, the computation of

$$D(-,-) = A(-,-) + B(-,-) * C(-,-)$$

proceeds as if it had been coded

```
c   get temporary vector, say x.
    DO i = 1, total length of A/1024
      x(-) = B(-,i) * C(-,i)
      D(-,i) = A(-,i) + x(-)
```

That this approach is valid requires one to deal with the linear algebra. Note, for example, that such an approach cannot be taken for *non-* linear algorithms.

Parameter blocks-talking to the vector boards through occam I

Despite it all, we do, indeed, compile the vector expressions which are syntactically correct into working code. The requirements for coding are straightforward since Floating Point Systems provided a very complete set of primitive functions, called parameter block routines. These routines can be used for most situations (see [Figure 2](#)).

```

set up the skeleton parameter block
        pb.start(initializedblock)

stride through all but the final partial stride
        pb.continue

set up final partial stride
        pbfinish(VP.BLK2)

```

Figure 3. Vector parameter block routine skeleton

The standard skeleton for activating the vector processor is shown in Figure 3. The three routines explicitly mentioned communicate directly with the VPU, in effect, starting the vector operations. Figure 4 shows the complete occam I program segment for

$$x(-) = A(-) \times B(-) + s.$$

The parameter block routines have nine parameters. Several of the parameters contain subpieces which can be constructed dynamically. The first field is the function as the operation code. There are five separate subfields to be dealt with (Figure 4).

1. The ABmulSadd is the 'vector form' and serves to define the routine through the multiplier and adder.
2. The multiplier function (MF.mul).
3. The adder function (AF.add).
4. The precision of the operands (double precision, in this case).
5. Length in operand precision, with a maximum of 1024 total bytes.

The next fields deal with the location (bank) of the non-A-bank operand B1,

```

VP.BLK1[VP.opcode] := VF.ABmulSadd ∨
                    VP.func[MF.XYmul] ∨
                    VP.func[AF.XYadd] ∨ VP.d4 ∨ 128
VP.BLK1[VP.sload] := VF.Bloadl ∨ VP.64
VP.BLK1[VP.srcs] := BTEMPSCALAR
VP.BLK1[VP.srcv1] := 948224
VP.BLK1[VP.srcv2] := 737280
VP.BLK1 [VP.srcmsk] := VP. SV1V2
VP. BLK1[VP.dstmsk] := VP.nodst
        pb.start(VP.BLK1)
VP.BLK2[VP.opcode] := VF.ABadd ∨ VP. func[AF.XYadd] ∨
                    VP.d4 ∨ 128
VP. BLK2[VP.srcmsk] := VP.nosrc
VP. BLK2[VP.dstmsk] := VP.V1
VP,BLK2[VP.dstv1] := 947204
        pb.finish(VP.BLK2)

```

Figure 4. Occam I segment for vector process $x(-) = A(-) \times B(-) + s$

location of the scalar (BTEMPSCALAR). Next come the two operand locations, A at 948224 and B at 737280. The source and destination masks tell the VPU where and how many operands there are and the disposition of the result. Note that occam I does not have pointers; therefore, we had to either know the address or use a built-in function to get the address (see Figure 4).

The second call to the vector board is merely to store the results in main memory. This step is necessary only for final results, not intermediate ones, which are either held in the registers or saved in temporary memory locations. There is a *caveat*—the vector registers must be ‘refreshed’ frequently; therefore, these registers are not to be treated as temporaries. Again notice the absolute numbers for the location of the x vector.

With the vector boards, the result of any vector operation stays in the vector registers. The destination field tells the vector function where the result of the operation is to be stored. For any vector function call this destination field is the destination for the result of the previous call to the boards, not the current call. This makes it necessary to keep track of what is in the register and where it will be stored. While evaluating an expression, it is occasionally necessary to store the result of the previous operation in a temporary. Because the result is not stored in memory until another vector call is made, it is necessary to follow the content of the register. If it is stored in a temporary, and used later, the address of the temporary must be used as a source parameter in the vector block. In this respect, coding the vector functions does not differ greatly in concept from scalars. Of course, there is a much more extensive operation repertoire for vectors and larger optimization peepholes are needed.

In order to make vector function calls, then, a parameter block must be completed. Internally, we used a structure corresponding to the block, which is filled in by the various functions. Once the table is complete, one routine is called to produce the occam I code.

IMPLEMENTATION DETAILS

The implementation of the C to occam I preprocessor started in the compiler class taught by Stevenson. The remaining authors have participated through various incarnations of the preprocessor. Most techniques used were familiar to those who have taken a course in compiler implementation. However, the unique nature of the preprocessor made it necessary to push past the standard techniques to different techniques, which we now describe.

Pass I implementation

We used—for C—the System V definition used by Portable compiler.³ This definition is for *yacc*.⁴ The functional attributes which are part of the portable definition were replaced by output routines which wrote intermediate code to a file. Our intermediate code was a string of numbers as described below. The Fortran definition was developed from scratch using VAX Fortran and the Fortran 77 standard as a guide. The two languages were modified to accommodate the syntactic changes described above.

The scanner for C was constructed using *lex* even though a hand-coded scanner

is certainly preferable. * The block structured symbol table was constructed using the standard techniques. The scanner for Fortran was hand-coded.

Pass II implementation

The input to the second pass is a string of numbers. These numbers represented an index into an array of context definition records, called *syntabs*. Every unique symbol had an entry. The first pass disambiguated the scoping of names through this method. Therefore, the second pass had all the syntactic structure for constructs but not for scoping. *yacc* was used to control input and to drive processing.¹²

Our approach to generating code was to treat the machine dependent optimization and assembly as a separate 'compiler', termed here 'pass II'. We asked ourselves, for example, 'what does an if statement look like in occam I?' and then tried to translate the C if statement to an occam I if statement. The upshot of this was we did not convert expressions into the standard intermediate language forms (say triples), but translated them into occam I statements.

In order to deal with the C type conversions, we used the *syntab* structures to keep track of the type of the expression being evaluated. If an integer were multiplied by a float, for instance, we would generate an occam I call to convert the integer to a float, then do the multiplication. The type of the expression would then become a float. If the result were assigned to an integer we would then know to generate another occam I call to a conversion routine.

This approach lets us support the C idea of true-false: 0 is false, everything else is true. In C the expression if (A = B) is legal; in occam I it is illegal. In pass II the expression A = B would have a type of integer when we were expecting a boolean. We would then know to generate an occam I statement assigning B to A and the if statement would then test whether A was equal to 0.

We have three stacks in pass II: the *pre-* stack, the *stk-* stack and the *post-* stack. In order to support Cisms like ++A and A++ we keep track of statements that must be generated before an expression, and statements that must be generated after an expression. Calls to conversion routines, for instance, need to be generated before the expression being translated is generated while ++ could be either.

The choice of *yacc* control in pass II processing was not altogether satisfactory. There was difficulty in knowing what point in the grammar is being parsed. A complicated system of flags is used whose sole purpose is to deal with nested statements. It might have been possible to rework the grammar; however, by the time the problems became apparent it was easier to continue with the flag solution. Such undesirable techniques might have been avoided if we had available a top-down parser generator. However, the *yacc* method is quite usable and well studied.¹²

One point of interest in vector processing is the need to deal with 'triplets'. A triplet is three operands—two of which are vectors—and a scalar combined by two operators. For example, the SAXPY operation in LINPACK is such an operation with type

scalar × vector + vector.

These forms are the most natural way to represent vector expressions. The triples

* A major teaching objective of the course was an emphasis on took

serve as the point of organization around which the myriad of rules of the VPU can be interpreted. In ordinary expressions, one typically has only one operator and two operands to worry about simultaneously. When such an ordinary operator is parsed, two operands, previously pushed on the stack, are popped and code can be generated. In the case of vector expressions for our VPU, it is possible to optimize using the triplets. Instead of generating code immediately, a vector block is prepared and pushed onto the stack. If, as the parse continues, there is no possibility of making a triplet call, the triplet is discarded and the more conventional binary/unary operators are generated. If, however, a triplet can be generated, there is enough information stored in the vector block to generate code.

The goto problem as solved in the Fortran compiler

The problem with converting programs with gotos into programs with just if-then-else and whiles is certainly a familiar one. For most programmers, the solution is simply not to think about gotos at all. However, one purpose of the Fortran compiler was to get up and running on the T-series as soon as possible. This meant compiling older Fortran programs. (By way of contrast, the C compiler was used for graduate level classes, so the problem was not as severe here.)

Since the proof that every program with gotos can be rewritten without them is constructive and well-known, one could expect to transform programs into occam I without too much trouble. Our approach was to have pass I transform the program; this simplified pass II. The basic approach was to read in the entire Fortran program and form a list version. From this list version, we transformed the program into the 'extended basic block' format.⁴ Using the basic blocks, the 'goto' graph was formed, with the 'nodes' as sequential code having no gotos and all of the gotos at the end. That is, in *gotoful*- style, as shown in Figure 5.

```

label: statement,
      statement,
      ...
      statement
      if( condition1 ) goto label,
      ...
      if( conditionn-1 ) goto labeln-1
      goto labeln

```

Figure 5. Form of an extended block

In this rearranged form, it is easy to convert the program into the while form by the following algorithm. Suppose there are a total of $k + 1$ blocks in the program, numbered from 0 to k . Let the zeroth block be the 'prologue' of the procedure. We can also associate a distinct number with each label. Then the block form of the program can be encapsulated in a while statement of the form shown in Figure 6. As a simple illustration of this technique, we take the following C program for finding the maximum of an integer vector and convert it to the style in Figure 6. The program might be coded as in Figure 7 (certainly only an illustration). The equivalent goto-less program is shown in Figure 8.

```

WHILE tempname <>0
...
IF
    tempname = i
    Code for Block i in the form shown in Figure 5
...

```

Figure 6. Skeleton form for goto programs

```

integer maxvec( X, N)
integer X(N)
integer I
I = 1
maxvec = 1
lif(I.gt. N)return
if ( X(I) le. X(maxvec) ) goto 2
maxvec = I
2 I = I + 1
goto 1
end

```

Figure 7. Example goto program

CRITIQUE OF OCCAM I PROBLEMS

This section summarizes the problems we encountered in the project. Once again, we emphasize that these criticisms are only considered valid when attempting to preprocess into occam I. We certainly do not intend them as criticisms of occam I when used by programmers nor is this a criticism of occam II.

Obstacles to compiling sequential programs into occam I

1. Occam I requires indentation to indicate program structure. During preprocessing, one often produces many more blocks than a programmer does. Often, then, the output of the compiler was unreadable without considerable editing of very large files.
2. Occam I only supports one dimensional arrays. This is no great problem for relatively small programs produced in classes. For larger projects and scientific codes such a restriction is unacceptable.
3. Passing parameters is always by reference when using arrays. This leads to several problems, in particular the creation of temporaries for every constant appearing as an actual argument. Another problem created by the array handling has to do with passing a virtual origin; i.e. passing only part of a vector. This is forbidden. Our alternative was to pass a *dope vector* containing the real and virtual origin. This considerably complicates preprocessing.
4. Occam I does not have a goto statement! Therefore, translation of C and Fortran programs with said construct must be reformed. The principles for this transformation are easy enough, but it was a lot of work.
5. Several lines of code must be produced just to write one line of code to the

```

PROC maximum( VALUE integer X[], N, Return)
var integer Max, TempX[]:
seq
var integer tempname:
tempname:= 0
while tempname >= 0
if
tempname = 0
seq
Temp:= 0
Max := 0
tempname:= 1
if
tempname = 1
if Temp = N
seq
tempname:= -1
Return:= Max
if Temp <> N
tempname := 2
if
tempname = 2
#X[Temp] > X[Max]
seq
Max := Temp
tempname := 3
if X[Temp] <= X[Max]
tempname:= 3
if
tempname = 3
seq
Temp:= Temp+1
tempname:= 1

```

Figure 8. 'Maximum' program without goto s

output file. Part of the cause for this is that occam I does not have operator precedence rules. Therefore, we were forced to (conservatively) parenthesize every expression. It might have been possible to produce a 'prettyprinter' but this was not seen as justified.

6. Occam I does not support the use of functions—everything must be written as a procedure. For C that means every procedure has hidden arguments: one for the any result which might be returned. Variable numbers of arguments are forbidden by occam I. As it stands, neither preprocessor can enforce this restriction.
7. The if statement in occam I does not behave like most other if statements do. However, if one thinks of it as a switch or case, the if statement is quite usable.
8. Occam I does not support recursion which leads to limited programming or the addition of stacks to provide for programmer-written recursion. It is inconvenient for some semantics.
9. Occam I does not have a native memory allocation mechanism for space

management. This leads to Algol style temporary allocations. Such a local mechanism is merely inconvenient for most situations, but causes problems creating global items.

10. Because occam I does not allow changes to variables passed by value and C does, we adopted the convention of defining formal parameters to a function definition with temporary names. Then, in the body of the function, we declare the user's original parameters as variables. The first statements in the function, then, become assignments of the temporary names, passed to the function, to the variables declared in the body. This effectively simulates call by value, and occam I does not complain.
11. In order to pass by reference, the formal parameter is declared as a pointer (VAR). In this case the users formal parameter is declared as a formal parameter in the occam I function.

Obstacles in using occam I when also using attached processors

The obstacles here are very fundamental: occam I simply does not have the capacity for dealing with interrupts, nor does TDS provide an assembler. By the time one observes all the restrictions in occam I and VPU, one has some of the most involuted code conceivable.

1. The T-series (using the T414) could not perform scalar floating point arithmetic directly. This low scalar performance has been a negative factor in the acceptance of the T-series.
2. For higher dimensional hypercubes, the T414 can only be connected to four at one time. The time to switch among the other T414s is unacceptably high.

If a processor wants to communicate with another processor that is not directly connected to it, then the programmer must provide the appropriate path for the data to be passed along. Due to the synchronous communication, the program must be written so that there is a minimum amount of time between the data being sent by one processor and the data being asked for by the receiving processor. Such machinations are difficult in occam I—let alone in a preprocessor.

SOME FINAL THOUGHTS-A PROFESSORS PERSPECTIVE

The exercise discussed here was extremely fruitful for the participants. The graduate students were able to make a real contribution on a real problem and at the same time explore several new architectures; both the transputer and the T were completely new concepts when they started. One certainly cannot go to a standard text and pull the algorithms from the appendices. The exercise also gave these students a chance to learn a tremendous amount about non-shared memory compiling.

From a teacher's (Stevenson) perspective the above justifies the effort. However, an unexpected realization accrued. In the past, I have had students write preprocessors from a restricted language, say Modula-2 to C. These have been a moderate success and good teaching tools. This preprocessor was the first time I had tried C to a restricted language. Much of the emphasis on preprocessors comes from the belief that industrial applications for preprocessors make problem-oriented languages economically viable. A second benefit is to preserve dusty decks by 'upgrading'

them through preprocessing. Our experience here would suggest otherwise. As we emphasize the creation of new systems using new languages, we are also forcing the hand-translation of older, economically important, systems.

The harness concept is a very viable approach to parallel processing *if the standard compilers are available*. In retrospect, trying to force C and Fortran to accept as their virtual machine something which they were never meant to have was the root of our problems.

Ultimately, our bypassing the development of low-level targets for preprocessing may be a mistake. We certainly support the designing of languages which encourage good programming practices. But one must not lose sight of the tremendous base of software which exists in the world and the consequences to advanced development from an intransigent stand on supporting the upgrading of these older systems.

It is clear from our experiences that the ultimate acceptance of parallel processing is still a people problem: how to take the programmers of today and re-educate them while maintaining, uninterrupted, the flow of software products. Language will enhance or retard that movement.

ACKNOWLEDGEMENTS

Office of Naval Research funding of the Clemson University Research Initiative bought half of the T-20. The other half was funded through the Division of Computer and Information Technology at Clemson, headed by Dr. C. J. Duckenfield. DCIT also funded development, working through Mr. C. C. Heck. The Fortran effort is in collaboration with Dr. D. D. Warner of the Department of Mathematical Sciences at Clemson. Floating Point Systems has been quite helpful; we wish to thank Martin Waugh (now at Intel). We would especially like to thank the two referees for their time and patience. This is a much better paper for their efforts.

REFERENCES

1. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
2. Robin Milner, *A Calculus of Communicating Systems*, (Lecture Notes in Computer Science, Number 92), Springer-Verlag, New York, 1980.
3. S. C. Johnson, 'A portable compiler: theory and practice', *Proc 5th ACM Symp on Print. of Prog. Lang.*, 1978.
4. Alfred V. Aho, Ravi Sethi and J. D. Unman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.
5. INMOS Limited, *Occam I Programming Manual*, Prentice-Hall International Series in Computer Science, C. A. R. Hoare, (ed), Prentice-Hall International, Englewood Cliffs, NJ, 1984.
6. Roy Dowsing, *Introduction to Concurrency Using occam*, Van Nostrand, Reinhold, London, 1988.
7. R. Poutain and D. May, *A Tutorial Introduction to occam Programming*, Blackwell, Oxford, 1987.
8. Karen A. Frankel, 'Evaluating two massively parallel machines'; *CACM*, **29**, (8), 752-758 (1986).
9. William A. Wulf, D. B. Russel and A. Nico Habermann, 'Bliss: a language for systems programming', *CACM*, **14**, (12), 780-790 (1971).
10. M. Milenkovic, *Operating Systems: Concepts and Design*, McGraw-Hill, New York, 1987.
11. E. W. Dijkstra, 'Guarded commands, non-determinacy and the formal derivation of programs', *CACM*, **18**, (8), 453-457 (1975).
12. R. Steven Glanville and Susan L. Graham, 'A new method for compiler code generation', *Proc. of 5th ACM Symp. of Prog. Lang.*, 1978.