NASA Contractor Report 181691

ICASE INTERIM REPORT 5

PROGRAMMING THE NAVIER-STOKES COMPUTER: AN ABSTRACT MACHINE MODEL AND A VISUAL EDITOR

David Middleton, Tom Crockett, and Sherry Tomboulian

NASA Contract No. NAS1-18107
August 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographies, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

Programming the Navier-Stokes Computer: An abstract machine model and a Visual Editor.

David Middleton, Tom Crockett and Sherry Tomboulian

ICASE, NASA Langley Research Center

The Navier-Stokes Computer (NSC) is intended to apply large numbers of floating point ALUs to computational problems that can be expressed using calculations on long vectors [Nosenchuck *et al.* 87, Tomboulian *et al.* 88]. Since programming language considerations were ignored in the hardware design, efficient operation will depend on machine level programming. The Visual Editor developed here is intended to provide a supportive environment in which the programmer can more effectively write machine language programs.

Any programming system presents the user with a model of computation. The abstract Navier-Stokes computer described in this document is an explicitly chosen model for the Visual Editor to present. We prefer this approach to having the computational model evolve implicitly while the Editor is constructed. We do not use the complete NSC as this model for several reasons, in particular because of its complexity and, as yet, lack of stable definition.

The abstract model is a subset of one node in the actual machine. This allows us to ignore issues of synchronisation, communication and multiprogramming that arise in the actual machine and to avoid implementing features not provided directly by the hardware. Naturally, it is hoped that the abstract model presented would waste little of the computational power of the eventual NSC (at least, for many problems) and that the Editor would provide trapdoors to allow the programmer to use the ignored features (although possibly only with relative difficulty).

The basic philosophy of the Visual Editor is to provide support and verification to the programmer building the complex microcode structures since even the abstract model remains ill-adapted to compilation tools. That is, the programmer makes all programming decisions, in particular, those regarding the allocation of resources; the Editor merely indicates errors without suggesting alternatives. A method for programming is developed (from which the Visual Editor's operations are derived) which we hope will simplify the programmer's task.

This paper deals with three separate things which must be kept distinct: the actual node with its abilities, the abstract, subset node with its abilities, and the programming method with the attendant verification and abstractions to be provided by the Editor. The first section describes the abstract hardware model; the second section describes the programming process and the way that the Visual Editor would support it. The paper assumes fairly detailed knowledge of the "full node" of the NSC. Since its design is still being completed, the level of detail being attempted here will naturally lead to some inaccuracies.

Abstract Navier-Stokes Machine node: a subset of reality.

An abstract node has three parts: computation units which are connected to form *pipes*, *pipelines* consisting of pipes and a memory system with DMA controllers which assembles and feeds vectors into them and store vectors of results, and a central controller which (statically) schedules pipelines.

Computation Units.

These comprise 2 shift/delay units and 32 ALU's. The shift/delay units allow a vector arriving from memory to be duplicated with different offsets. Each one contains four serially connected FIFO queues whose outputs are available to *some* ALU's through the local switches.

Each ALU has a small internal register set which can be used in several ways, such as supplying the constants of an expression being evaluated. Each ALU is symmetrical, that is, if it can do $A \odot B$, it can do $B \odot A$. In the actual node, this may involve switching inputs or manipulating the ALU's output. Each input has a delay unit, called a vector latch, for aligning the ALU's vector operands. The delay period can vary from zero to the size of the register set. The ALU's are *not* homogeneous with respect to the functions they can perform; there are at least three different kinds of ALU depending on whether integer and logic or minimum and maximum operations are available.

The possible connections between ALU's are restricted in order to increase the number available. In particular, the 32 ALU's are hardwired into 16 Arithmetic/Logic Unit Structures (ALS's) as 4 singlets, 8 doublets and 4 triplets. Programs can only control the output destination of the 16 ALU's at the outputs of the ALS's.

Implementation aspects.

Actual ALU's can be internally configured in various ways; these correspond to specific uses, such as evaluating recurrences, which can be displayed directly in the high level representation of the abstract ALU's in the Editor.

For connecting various units together, an actual node contains caches and switches (labeled M×F, D×F, D×S, F×F, S×F, F×D), both of which are absent from the abstract model. In the transition from the model to an actual node, the caches can be allocated in a straight-forward fashion, since the ability to store information between pipelines (by using the caches as vector registers) is ignored. The implementation of the switches leads to some complicated restrictions that will be explained in the programming section.

Memory System.

Memory in the abstract node consists of 16 planes (each with 128 mega-words) with hard boundaries between them; specifically, the user must allocate storage for variables so

that the inputs and outputs of each *pipeline* reside in individual disjoint memory planes. This exists because in the actual node, significant difficulties or penalties occur if, in a single pipeline, one variable spans multiple planes or one plane holds multiple variables.

In the actual node, 16 Pipeline DMA units (PDMA's) must be programmed to generate and feed vector streams into pipes or to store vector streams back in memory. The corresponding input and output blocks in the Editor are given vector specifications which consist of an *initial address*, a *count*, a *stride*, a *repetition factor* and a *pipeline delay*. The first three are obvious; the repetition factor allows a single input value to be repeated or several output values to be overlain (thus keeping only the last one); the pipeline delay indicates the number of extra values which must be sent at the end or stored in front of the actual vector due to the filling and flushing a pipeline requires. The Editor ought to be able to deduce vector specifications from information provided during the programming process.

Implementation aspects.

The facilities provided by the caches and their controllers and the pipeline and memory DMA units are used only to implement the vector specifications described above. The switches in the actual memory interface (M×D, D×M, M×M, F×M) are absent from the abstract node, and in at least two cases, some loophole specification method is necessary. The M×M switch would likely be used in cases where a variable needs to reside in different memory planes for different pipelines. Also, communication between different nodes (through "hyperspace") uses these ignored facilities.

Assuming cache ordering is irrelevant, in each pipeline we arbitrarily number the input caches 1 to n and the output caches n+1 to n+k (where $n+k \leq 16$). We assume caches are initially empty and are flushed after each pipeline finishes, that is, they do not hold results to be used from previous computations. This appears to be optional in recent versions of the actual machine (each cache has a 'sticky' bit, called "read/write").

For the actual node, several independent pipelines (built from disjoint resources) can be separately or jointly initiated to operate in overlapping time periods. The definition of a *pipeline* in the abstract node is now extended to include several disjoint pipelines as defined earlier. The extension merely allows a pipeline to have disconnected parts. The difference from the actual NSC is that these component pipelines may not be dynamically scheduled: a set of component pipelines that may operate together are statically scheduled by the programmer always to run together. The components would operate in lockstep except that individual cache misses might stall one component while others proceed. Each component pipeline is internally synchronous; a delay at any of its PDMA's stalls all the PDMA's in that component. Although cache misses should in fact be predictable (the machine is not multiprogrammed), we view cache misses as indeterminate for simplicity.

For each component pipeline, one PDMA is designated to send an End-Of-Pipeline (EOP) interrupt to the central controller when that pipeline finishes. The implementation of central control would be responsible for awaiting several PDMA EOP interrupts, one from the distinguished PDMA in each component, before initiating the next extended pipeline.

Flow of control

In an actual node, the central controller initiates pipelines by issuing the appropriate long instruction words to the various units described above. (Individual fields can be disabled so as not to interfere with other pipelines already in progress). It contains a micro-sequencer which selects the appropriate pipelines by executing microcode in a conventional way. We ignore the ability in the actual node of the Pipeline Status Table (and several other hardwired processes) to initiate pipelines. If this facility cannot be disabled, then the pipelines issued in those cases should have all their fields disabled.

In the abstract node, a pipeline will generate as many interrupts before it is finished as there are disjoint component pipelines. Having initiated a pipeline, the main microsequencer waits for this number of interrupts before proceeding to its next instruction (which may be to initiate another pipeline or to execute its own code implementing more complex control flow).

For the Visual Editor, a simple standard block-oriented language could be built on top of this machine model. The basic statements, pipeline initiations, would be aggregated using looping and conditional statements. Some loophole might be necessary to allow specifying blocks of micro-code, particularly, for example, in implementing the logical tests associated with control flow statements.

Conditional Vector Expressions

The actual node provides a mechanism for performing some conditional computations at full speed inside vectorised loops. This includes merging two vectors according to a logical mask, a feature which is available in some commercial vector machines. Examples (in the language C) that can be implemented as Conditional Vector Expressions are "`for(i=a;i<b;i+=c) { if (test(i)) D[i]=f(i); else E[i]=g(i); }`" and "`for(i=a;i<b;i+=c) D[i] = ((test(i)) ? f(i) : g(i)) ;`".

These expressions can be implemented as single pipelines with support from specialised hardware in the central controller, called the *condition code resolution* circuitry. The facility will be described by showing how the second example would be executed.

First, the functions *test*, *f* and *g* are implemented with three component pipes, using the appropriate input vector variables. Although not necessarily evident from any diagram, these pipes need to be linked so that if one PDMA unit, say in *f*, suffers a cache miss, then the others, including those in the pipelines for *g* and *test*, will also stall. Second, the memory specifications for feeding these pipes are organised so that the values *f(i)* and *g(i)*

are available exactly one clock cycle after the function *test(i)* has generated an appropriate flag for the condition code resolution circuitry. Finally, two versions of this aggregate pipe are created, differing only in that the output in one comes from the *f* pipe while that of the other comes from the *g* pipe. The condition code resolution circuit issues one of these two pipeline specifications every clock cycle, according to the value of *test*. As a result, either the output of the *f* pipe or the *g* pipe is sent to the variable D in memory. The actual node must not allow this continual issuing of pipeline instructions to interfere with, for example, the counters operating in the PIDMA units.

More details cannot be given about the abstract node's abilities as regards Conditional Vector Expressions until the actual node is better defined. Obviously, large numbers of function units, easily exceeding the number available, may be used (inefficiently) to maintain full vector speed in this fashion. In such cases, precomputing and storing any of the three vectors is an obvious alternative for the programmer to use.

Neglected abilities of the actual NSC node

This section is an incomplete in-line appendix listing facilities of the actual node that have not been exploited in the subset abstract node.

Caches contain 'sticky' bits allowing data to be stored from one pipeline to another.

The Pipeline Status Table can autonomously initiate pipelines following others' terminations.

The microcode is tree-structured rather than being flat, enabling specifications of memory configuration and pipe structure to be re-used. (This is due to the actual hierarchy not matching the abstract node's divisions which strongly interferes with any useful sharing of specifications).

Programming the abstract node with the Visual Editor.

This section describes two separate concepts: the programming process for the NSC and as a derivative, the interface and support functions to be provided by the Visual Editor. A basic motivation for the Visual Editor is that the complexity of the NSC will prevent, at least in the near future, the development of compilers that can produce adequately efficient code. The goal of the Editor is to provide the user with a support tool for machine level programming that presents decisions in a suitable order to minimise undoing previous choices, and that validates those choices as they are made.

The programming process is first described and then demonstrated using two examples (expressions to be evaluated within appropriate loops):

$$R_{i,j,k} = \frac{1}{h^2}(U_{i-1,j,k} + U_{i+1,j,k} + U_{i,j-1,k} + U_{i,j+1,k} + U_{i,j,k-1} + U_{i,j,k+1} - 6U_{i,j,k}) - G_{i,j,k}$$

taken from the paper by Nosenchuck, Krist and Zang, and later, $X_i = Z_i(Y_i - X_{i-1})$, a recurrence taken from the Livermore Loops [McMahon 86].

The programming process.

The Visual Editor presents a main panel for displaying a pipeline computation flanked by side panels which display a menu, variable declarations and the control flow program (see Figure 1).

The first stage of programming.

First, the programmer writes the expression occurring within the loop in a comment area at the top of the main panel and second, derives (on paper) a data flow diagram which implements that expression. This step is difficult to automate due on the one hand to the wide range of variations that can arise, for example, through applying associativity and commutativity and selecting common subexpressions, and on the other hand to the timing constraints which the machine imposes. These aspects are particularly evident in the linear recurrence example.

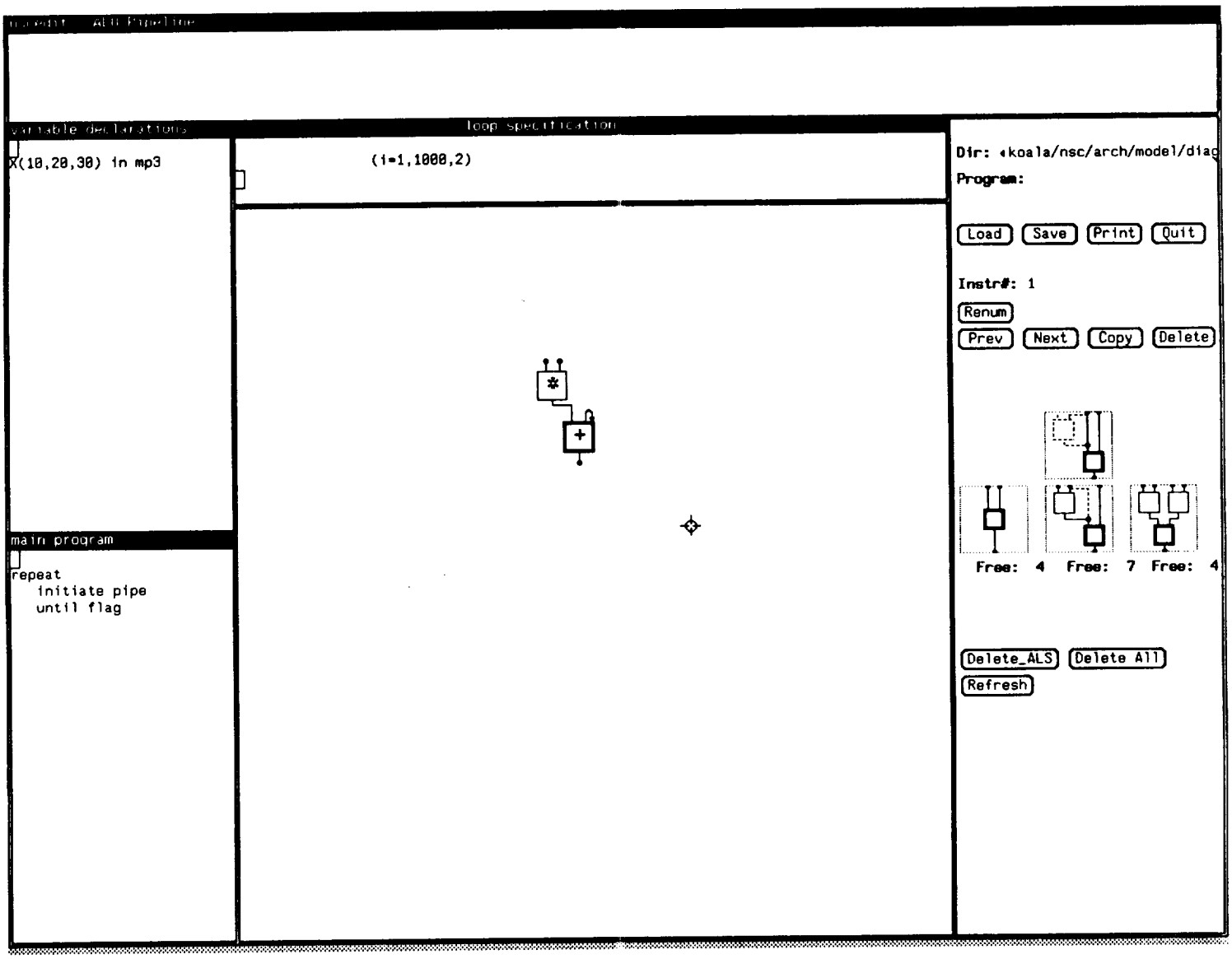

Figure 1. Visual Editor layout

Third, the user constructs a pipe corresponding to this data flow diagram, allocating singlets, doublets, triplets and shift/delay units from the menu. This step also includes many choices which affect path lengths. Path lengths turn out to be crucial in the linear

recurrence example. Restrictions on the operations available in different ALU's also manifest themselves in this step. Appropriate techniques to support automation include using back-tracking languages such as Prolog and using graph grammars with the ALS's (and the types of function units) as the terminal symbols.

Along with the ALS's and shift/delay units, the menu contains input and output blocks which generate (most of) the vector specifications for streams moving to and from memory planes.

The input and output blocks in a given pipeline *must specify distinct variables*. When one variable occurs several times in a pipeline (likely with different subscripts), all uses must connect to the same input block. In the pipeline, the different subscripts are implemented by different initial delays; this aligns the vectors entering computational units. The different delays can be seen with the variable U in the first example.

The first stage of analysis and validation.

Fourth, the Editor verifies that the different path lengths will correctly align the different vectors in the computation. Starting at each output block, it counts the delay from various points in the pipe, moving back towards the input blocks*.

Mismatches in path lengths can arise when two backward accumulations from output blocks arrive at one point in the pipeline. This occurs when a value (either an input or a local subexpression) is used more than once. In some cases, this can be remedied by choosing a non-zero initial delay at one of the output blocks; this non-zero delay would be incorporated into the vector specification created for the PDMA associated with that block. This approach of using the PDMA's ability to pad input vectors with leading and

* It counts one clock per ALU, one clock per connection (corresponding to traversing a Type 1 or 2 switch), and whatever delays were specified by the programmer for vector latches and shift/delay registers. The connections to and from a shift/delay unit together cause only one clock of delay. There are further delays associated with cache and memory connections, which are not yet completely specified.

trailing dummy values which output PDMA's will discard, will not work if a common subexpression is used multiple times in calculating the same result. In such cases, an error is signalled and the programmer inserts delays explicitly by using shift/delay units (a scarce resource) or by using vector latching in the ALU's along the short paths.

In the case of a duplicated input variable, the various path lengths leading to the input block must differ in the same amounts that the subscripts associated with each path differ.

The final value(s) for the path length of the pipe gives the pipeline delay, which is used later in creating vector specifications for the PDMA's.

When a value is used more than once, some complex restrictions in the switch networks will also occasionally cause difficulties. The F×F switch connecting the 16 ALS outputs to 48 ALS inputs is built from three 16×16 permutation networks, thus dividing the 48 ALS inputs into 3 disjoint sets, labeled *a*, *b* and *c*. When a single value is sent to multiple ALS inputs, those inputs must be distinctly labeled. This limits the fanout to at most three and further constrains which sets of inputs can share a value. The labelling details are shown in Figure 2.

We assume that simple numeric limits on fanout are sufficient at this stage and that the ability to swap inputs inside the ALS's can overcome the detailed restrictions, which the Editor checks at a later stage. The output of each unit is provided with several counts each corresponding to the fanouts of the switch blocks to which that unit provides inputs. For example, ALS's may be connected to other ALS's with a fanout of 3, to shift/delay units with a fanout of 1 or to output PDMA's with a fanout of 1 (via the F×F, F×S and F×D switches respectively). In the Editor, connecting an output block to a given ALS decrements the ALS's first count and connecting an ALS input to that ALS decrements its second count. Similarly, an input block has a fan-out of 3 to ALS's and 2 to the shift/delay units, and a shift/delay unit has a fan-out of 2 to ALS's. (Further, quite

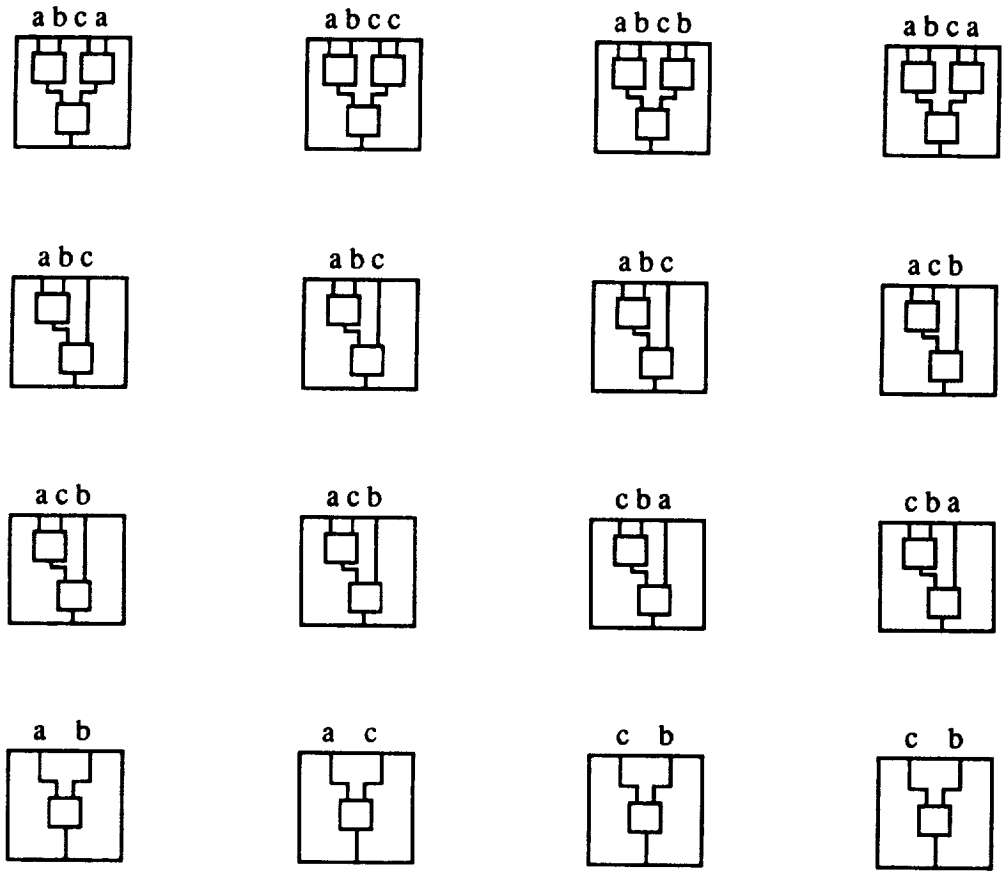

Figure 2. Labelling of ALS inputs

serious, restrictions apply to shift/delay outputs in that they can not be connected to all ALS's. This is dealt with later.)

A shift/delay unit displays four different delays each with respect to the input. The Visual Editor will check that the four serial queues which implement a shift/delay unit can support the given delays. The values are ordered and a limit of 8192 is placed on each of the three consecutive differences and the smallest value.

The second stage of programming.

Fifth, the programmer allocates specific ALS's to the singlets, doublets and triplets that form the pipe, satisfying the tightest restrictions first.

The greatest restriction on this step is that the outputs from the shift/delay units only connect to two triplets (ALS_0 and ALS_1), two doublets (ALS_4 and ALS_5) and one singlet (ALS_{12}). Furthermore, each of the eight outputs can only be used twice, once as an input to ALS_0, ALS_4 or ALS_{12}, and once as an input to ALS_1, ALS_5 or ALS_{12}. This restriction strongly affects the third step in which the programmer created the pipe from the data flow diagram.

The other difficulty lies with paths that fork, especially from the shift/delay units, but also from ALS's and input blocks. Sharing subgraphs, which causes such forking, has a large effect, even on the initial choice of data-flow graph, and so appears hard to automate. ALS's must be allocated so that input *labels* that connect to any given fork point differ. It is not yet evident whether this will be difficult for the programmer. Again, the backtracking facilities of Prolog naturally lend themselves to this operation.

Allocation of remaining ALS's can be done arbitrarily.

The third stage of programming.

Having constructed a pipe to combine streams of values, the programmer next specifies how to generate the streams from multidimensional arrays.

Sixth, the programmer provides a *controlling loop* whose index is used to generate vectors. The loop, modelled on FORTRAN implied loops, is specified at the top of the main panel near the textual comment describing the expression. (If the programmer creates an extended pipe which has independent components, then each one should have its own loop index, however all connected input/output blocks must use the same index).

Seventh, the programmer declares the variables in the side panel. A declaration looks like "`X(10,20,30) in mp3`" which means the array X with subscripts in the ranges (0..9,0..19,0..29) is stored in memory plane 3.

Eighth, for each input and output variable, the programmer creates a stream specification which allows values to be read from (or, in the case of output blocks, stored into) a sequence of the variable's locations that differ by a constant stride. A stream specification will not allow a variable to be scanned in a transposed order from that of its storage; each "column" of the transpose could be fed to a pipeline computation, but explicit looping in the control flow program would be needed for the "outer loop". (It is unclear whether PDMA's in the actual node allow access to variables by "nested loops").

A stream specification comprises a variable name with a list of *subscript expressions* which matches the subscripts in the variable declaration. Each is a linear expression in the loop index, and where it has a consistent meaning, may be used to cover several consecutive indices.

The abilities and restrictions of stream specifications are best demonstrated through some examples (assuming that X, as declared above, is stored with its leftmost subscript varying most rapidly). "`X(i)`" with a controlling loop of "`i=0,6000`" scans X in storage order. "`X(i)`" with a loop of "`i=0,6000,2`" scans X in storage order, only taking alternate values from the "rows", and is equivalent to "`X(2*i)`" with a loop of "`i=0,3000`". "`X(i)`" with a loop of "`i=0,6000,3`" might well be forbidden, or at least flagged, since the increment, 3, does not divide the first index range, 10. "`X(i)`" with a loop of "`i=0,200`" gives the first plane of X. "`X(i,1)`" with a loop of "`i=0,200`" gives the second plane of X. "`X(i)`" with a loop of "`i=0,400`" gives the first and second planes of X. "`X(1,i)`" with a loop of "`i=0,300`" gives a second plane of X at a different orientation. No stream specification can be made for the third orientation since it cannot be scanned with a single stride. "`X(i,i,i)`" with a loop of "`i=0,10`" yields a diagonal.

Vector specifications are constructed from the control loop, the stream specifications, the individual offsets associated with uses of input variables, and pipeline lengths.

The second stage of analysis and validation.

Ninth, the Editor verifies that the variables fit in memory as declared and that for each pipeline, all the variables are in distinct memory planes. The actual NSC design may require that space be left preceding output variables to contain initial garbage values generated by the pipe; this is expensive in the case of large strides.

The number of iterations given by the controlling loop, added to the pipeline's path length indicates the *count* to be specified to the PDMA's. Offsets associated with the input and output blocks indicate appropriate offsets to the variables' base addresses, yielding the *initial addresses* for the PDMA's.

Subscript expressions within the input and output blocks yield the *stride* for address specifications. Subscript expressions can be checked for out-of-range violations with respect to their variable's declaration.

It appears that the *replication* field, while useful for the linear recurrence example below, cannot be determined through this model of the Visual Editor. Explicit specification by the programmer will be necessary.

The fourth stage of programming.

Tenth, the user writes a control flow program to initiate pipeline computations in a language similar to assembly language and BASIC. This step cannot be designed in more detail until the actual NSC node is better defined.

Final notes.

From the subscript expressions and in particular the offsets, it ought to be possible to derive delays such as those needed to address neighbours in a grid, as is used in the first example.

For each pipeline, a particular Pipeline DMA unit is selected to notify the main sequencer when the pipeline completes. Any variable (input or output) seems sufficient for this task.

Distributing variables across memory planes is extremely useful if a pipe can be duplicated since this provides further parallelism. This is the responsibility of the user, and requires that variables be manually partitioned into separately named pieces.

The following appears to be a reasonable method for the Editor to calculate path lengths. Where possible, path length discrepancies are resolved by altering initial lengths at the output blocks so that all begin with non-negative values and at least one is zero; otherwise require the user to insert delays as described above. Calculate the list of path lengths for each use of an input variable and the list of offsets derived from the subscripts associated with each variable's use. Compute a third vector of non-negative values such that this vector plus (position-wise) the initial path length vector plus the offset vector gives a constant vector. This third vector represents the delays that must be inserted at the input PDMA's, and the repeated value in the constant vector is the pipeline delay, that is the number of clock cycles which the PDMA's must run beyond the length given by the controlling loop.

Examples programmed.

Example #1.

The expression and a corresponding data-flow graph for the first example are shown in Figure 3. This is only one graph chosen from many possibilities to satisfy high-level considerations; it is created by applying arithmetic rules, such as, in this case, translating $a-(b\times c)$ into $a+(-b\times c)$. Let us assume that U has dimension 500^3 and so U will (barely) fit in a single memory plane (128 Mwords). However, generating $U_{i,j,k-1}$ and $U_{i,j,k+1}$, which are separated by 500,000 locations, is, first, not possible with the shift/delay units and, second, would add this amount to the pipeline length, which is otherwise just over 1000.

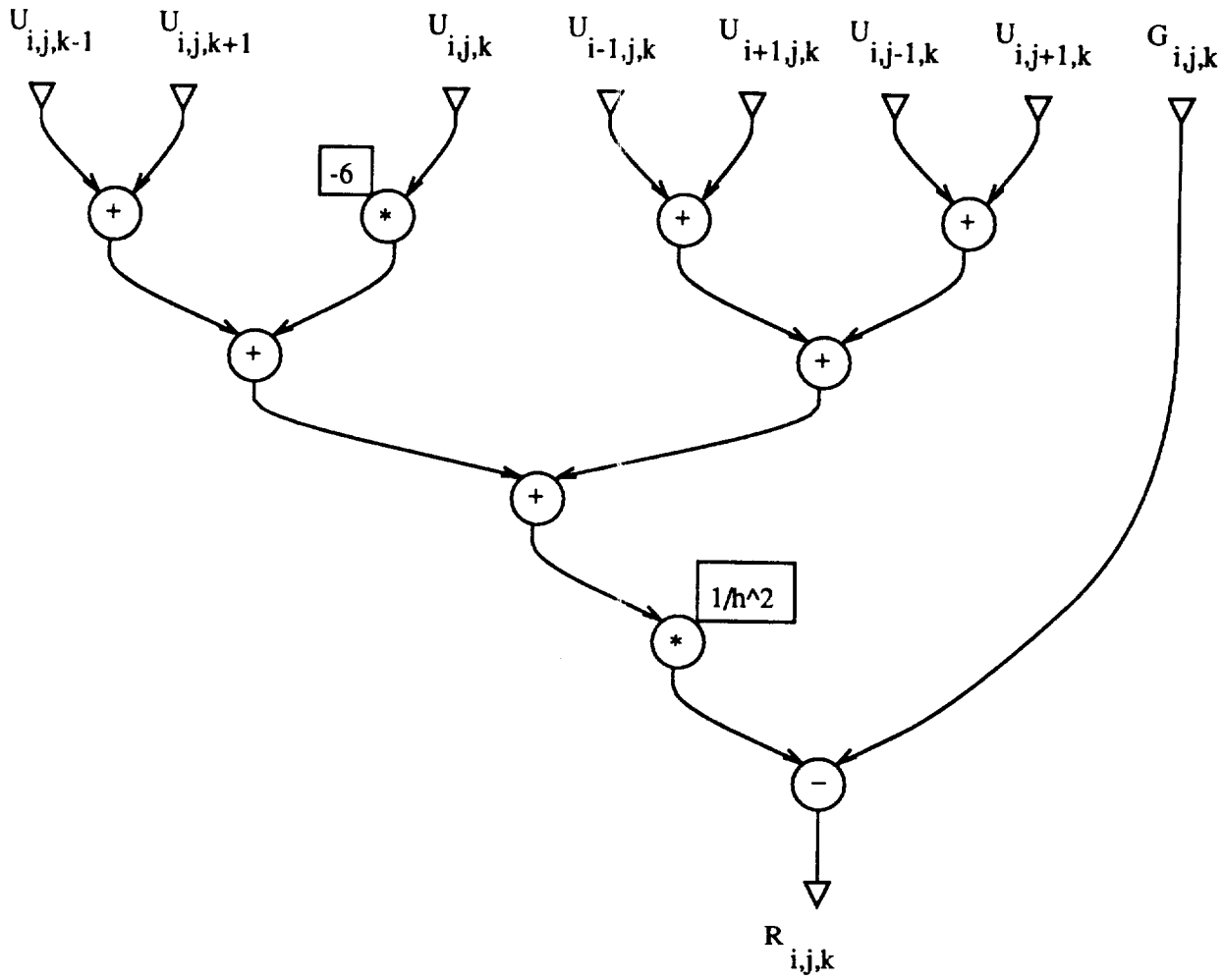

Figure 3. First example to be programmed

Hence, as described by Nosenchuck *et al.*, U is partitioned across three memory planes, according to the value of the third subscript modulo three, so that $U_{i,j,k-1}$, $U_{i,j,k+1}$ and the five values $U_{*,*,k}$, lie in distinct memory planes. (Since $R_{i,j,k}$ is used to update U, it may be independently necessary to create two copies of U in two separate sets of memory planes, alternately generating each from the other). The five values lying in the same plane, $U_{i,j,k}$, $U_{i-1,j,k}$, $U_{i+1,j,k}$, $U_{i,j-1,k}$ and $U_{i,j+1,k}$, are generated using a shift/delay unit from a single stream coming from the one memory plane.

A pipe created by allocating resources to this data-flow graph is shown in Figure 4. Various points demonstrate the decisions made at this point in programming. Since only 9 ALU's from the total of 32 are used, this pipe might be duplicated once or twice to increase the processing speed proportionally (requiring in turn that the variables U, G and R be further partitioned across separate memory planes appropriately). The number of memory planes, the lack of a third shift/delay unit and the restrictions on ALS's which can be connected to the shift/delay units prevent a third copy of the pipe from being built. Doubling the pipe probably requires dividing U into 6 memory planes to avoid performance penalties in accessing memory (although the actual PDMA's may allow interleaved access by two separate pipelines to the same memory plane).

The triplet and the upper doublet in Figure 4 must be chosen from ALS_0, ALS_1, ALS_4 and ALS_5, since only these can connect to the S×F switch. Restrictions on the shift/delay unit to ALS connections can affect the shape of the pipe. For example, the stream of $U_{i,j,k}$ values could be generated from $U_{i+1,j,k}$ with a one element vector latch in the same manner as $U_{i-1,j,k}$. Assuming that were the case, those three values could not be combined in a single ALS because of the constraints imposed by the S×F network; a single shift/delay output can be shared to create $U_{i+1,j,k}$ and $U_{i-1,j,k}$ only because of the reconfiguration provided inside ALU's.

Figure 5 shows the pipeline delays calculated at various points in the pipe, relative to the $R_{i,j,k}$ output which has been set to zero. Thick horizontal cuts across paths represent the delays of the switching networks. In order that the subscripted variables specified at the inputs will all meet correctly aligned, all the paths through the pipe must have the same length. The five values derived from variable U1, $U_{i,j,k}$, $U_{i-1,j,k}$, $U_{i+1,j,k}$, $U_{i,j-1,k}$ and $U_{i,j+1,k}$, have offset expressions of (0,0), (-1,0), (1,0), (0,-1) and (0,1), respectively, and require relative delays of 500, 501, 499, 1000 and 0 in order to be properly aligned. The lengths of their paths through ALS's are 9, 11, 9, 9 and 9, respectively, requiring additional

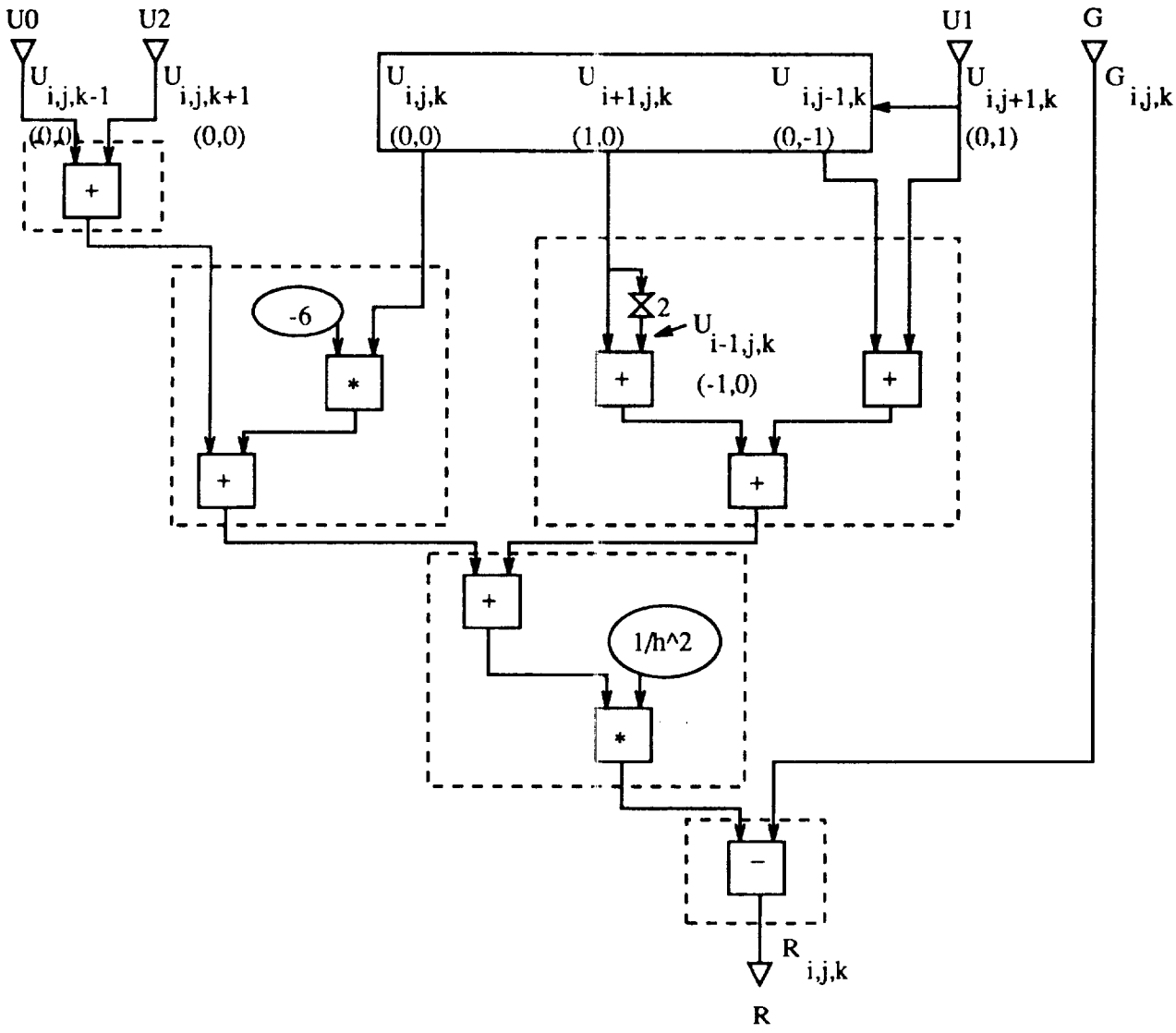

Figure 4. Pipe for first example

delays of 500, 499, 499, 1000 and 0 which are provided by the shift/delay unit. The Editor can derive these delays from the offset expressions, and the dimensions from the variable declaration, in this case "`U(500,500,500) in mp1`". The variables G, $U0$ and $U2$ with pipeline delays of 3, 10 and 10 also require additional alignment delays (1006, 999 and 999 respectively) which can be provided by the input PDMA's. The overall pipeline delay, the

time before valid data are being generated, is 1009, the delay through the pipe plus the longest shift in the shift/delay unit. PDMA's will transfer 251,009 values, the first 1009 of which the output PDMA's should discard.

Particular ALS's are now allocated to the pipe. The triplet must be ALS_0 or ALS_1; the upper doublet must be ALS_4 or ALS_5. The remaining six ALS's (assuming the displayed pipe is duplicated) can be assigned arbitrarily.

Variable declarations would look something like "`U0(500,500,167) in mp0`", "`U1(500,500,167) in mp1`", "`U2(500,500,167) in mp2`", "`U3(500,500,167) in mp3`", "`U4(500,500,167) in mp4`", "`U5(500,500,167) in mp5`", "`G(500,500,501) in mp6`", and "`R(500,500,501) in mp7`". It would be pleasant if a controlling loop something like "`i:0..500×500×167`" could perform one third of the computations throughout the volume U. However, since this pipe scans separated planes, G and R, as declared, cannot be scanned with a single stride. One possibility might be to store G as $G_{k,i,j}$, but that is obviously not a general solution to the problem. This leads to a control loop that looks something like "`i:0..500×500`" for a computation length of 250,000 data. A further problem lies in splitting G and R across two memory planes to feed two pipelines.

The control program would now be written to initiate these pipeline computations. First, there must be at least three pipelines, using $U0$, $U1$ and $U2$, respectively, as the source of the planes and the other two as the "vertical" neighbours. Further, it appears that the present NSC design would need 167 versions of each pipeline for the 167 starting addresses that correspond to different planes in the variables; no reasonable facility has been described in the NSC for providing run-time parameters to pipeline definitions (other than that provided through access to the microcode).

Comments on Example #1.

Examining this pipeline computation in detail shows several things. The complexities of the abstract machine and its real counterpart support the view that the Editor is merely

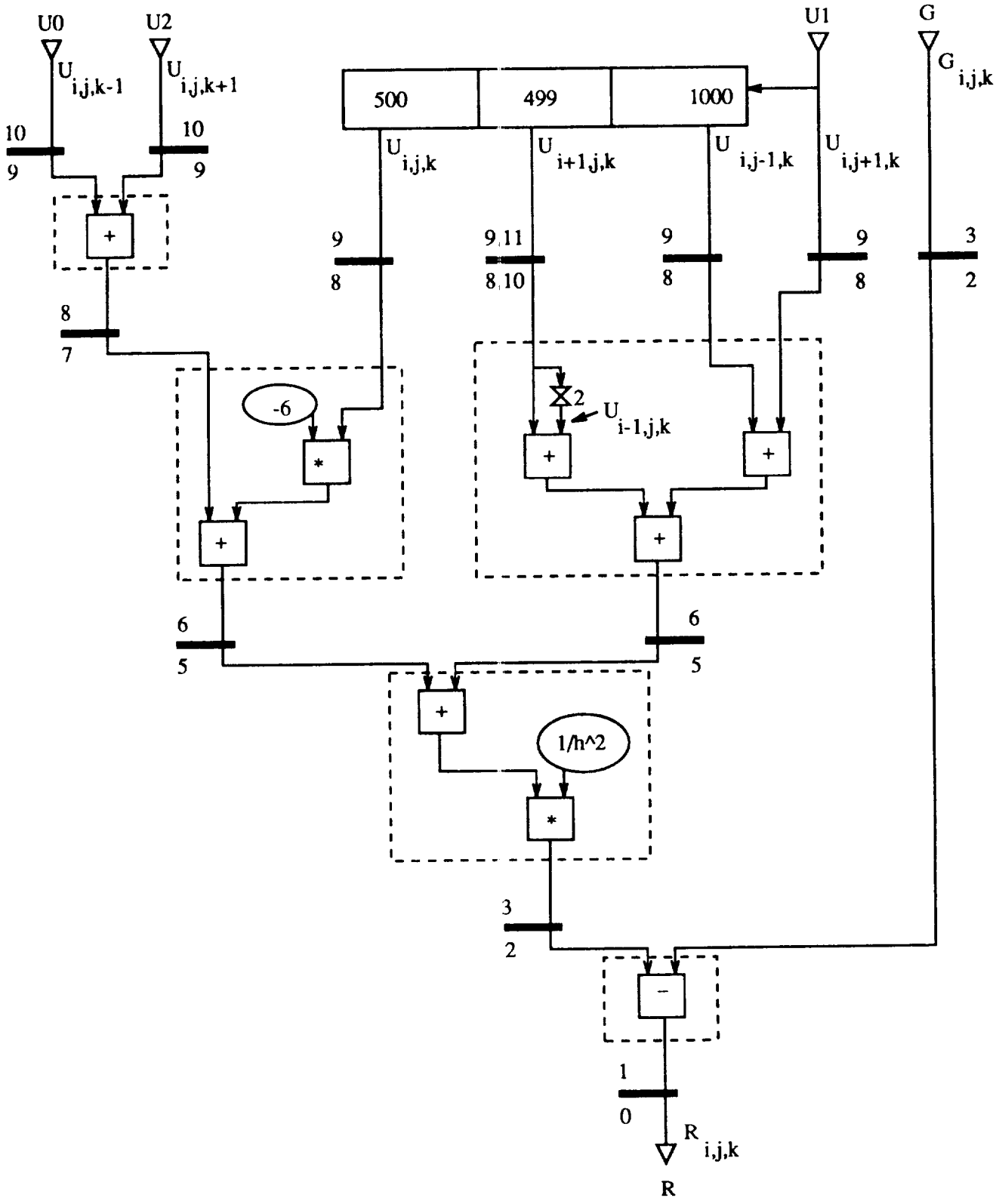

Figure 5. Pipeline delays for first example

an aid to user who must control all choices. The two principal steps appear to be determining the structure of the pipeline, and allocating variables to memory (which includes distributing variables among the planes and partitioning individual variables across multiple planes). It seems that the user must have a reasonably accurate idea for appropriate choices in these steps before the editing process begins; the programming process described above seems only useful for verifying and filling in the details of the final program.

This example also suggests that a major area for improvement with this machine is the memory system. The high computation rates seem to need a higher bandwidth than just sixteen paths to memory. Vector specification might well use a hierarchy of strides and counts to provide nested loops of access to variables, if the microcode controlling the DMA units could issue addresses rapidly enough.

Example #2.

In this example, we consider a recurrence taken from the Livermore Loops, $X_i = Z_i(Y_i - X_{i-1})$. Figure 6 shows a straight-forward implementation. Since there is a delay of three from the use of X_{i-1} to the availability of X_i at the same input, this pipeline relies on the *replication* facility in the vector specification hardware to discard the two intervening garbage values. This approach gives a vector rate of almost 7 MFLOPS, one third of the 20 MFLOPS provided by two ALU's, for a utilisation of 33%. There is a body of research on solving such first order recurrences in parallel; the following solution is used merely to study designing pipelines.

The above expression can be expanded to $X_i = Z_iY_i - Z_iZ_{i-1}Y_{i-1} + Z_iZ_{i-1}Z_{i-2}Y_{i-2} - Z_iZ_{i-1}Z_{i-2}X_{i-3}$, a first pipeline for which is shown in Figure 7. Several choices were made: the tree that combines the four terms to be added is deliberately unbalanced to minimise the feedback delay involving X. For this pipeline, the Editor ought to be able to associate the variable X_{i-3} with the appropriate ALS input in order to verify the delay specified. Figure 8 shows the path lengths relative to the output, as they exist before delays are

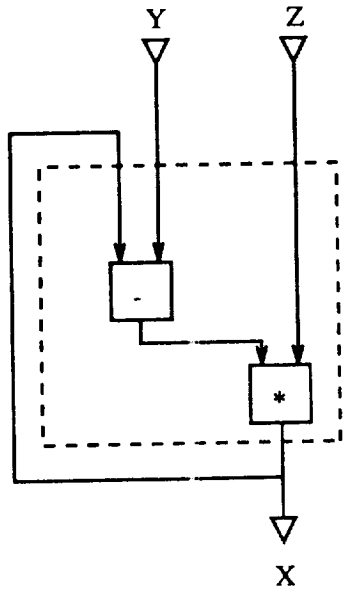

Figure 6. Simple recurrence

added. The branch labelled F has two paths to the output, but since both are of length 8, no action is necessary. The branch labelled G, however, leads to two paths of different lengths, initially 4 and 6. In response to an error message, the user would insert a two clock delay to correct this discrepancy; one possibility, shown in the diagram, is to add a two cycle vector latch in the lower doublet ALS. Although the branch labelled H leads to two paths of different lengths, the output path of length 1 and the feedback path of length 4, the Editor should not signal an error since the subscripts for the variables associated with the paths, X_i and X_{i-3} differ by the same amount.

Next, delays are added to align the input variables, which will incidentally resolve discrepant path lengths arising at the two remaining branches. From left to right, the variables and their pipeline lengths are: Z_i with 8, Y_i with 8, Y_{i-1} with 8, Y_{i-2} with 6, Z_{i-1} with 10, Z_i with 10 and Z_{i-2} with 8. Adding these lengths to the corresponding

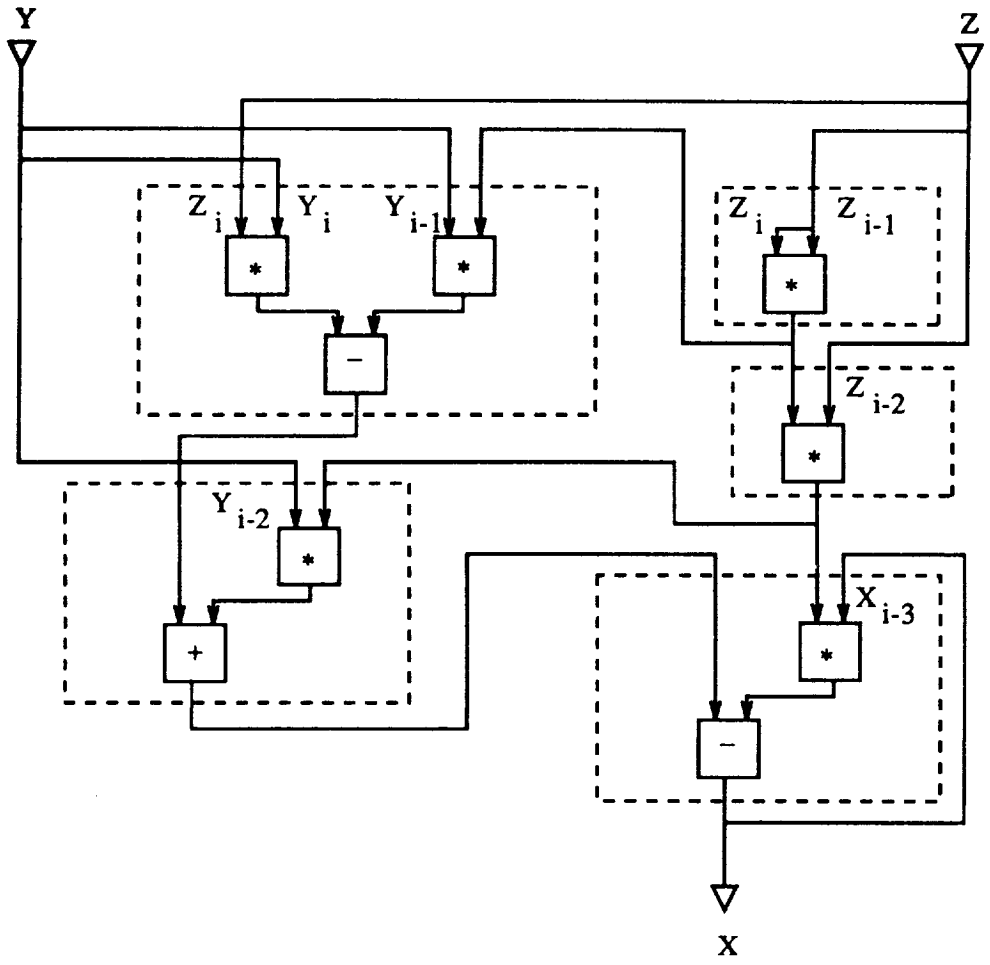

Figure 7. Recurrence restructured

subscript offsets yields 8 for Z_i, 8 for Y_i, 7 for Y_{i-1}, 4 for Y_{i-2}, 9 for Z_{i-1}, 10 for Z_i and 6 for Z_{i-2}. The difference between this list and a constant list of all 10's (the maximum value) is 2,2,3,6,1,0,4. The remaining six vector latches shown in Figure 8 provide these delays. (Note that the two two-clock delays in the triplet might as well have been combined into a single two-clock delay in the following ALU (the one doing a subtraction)).

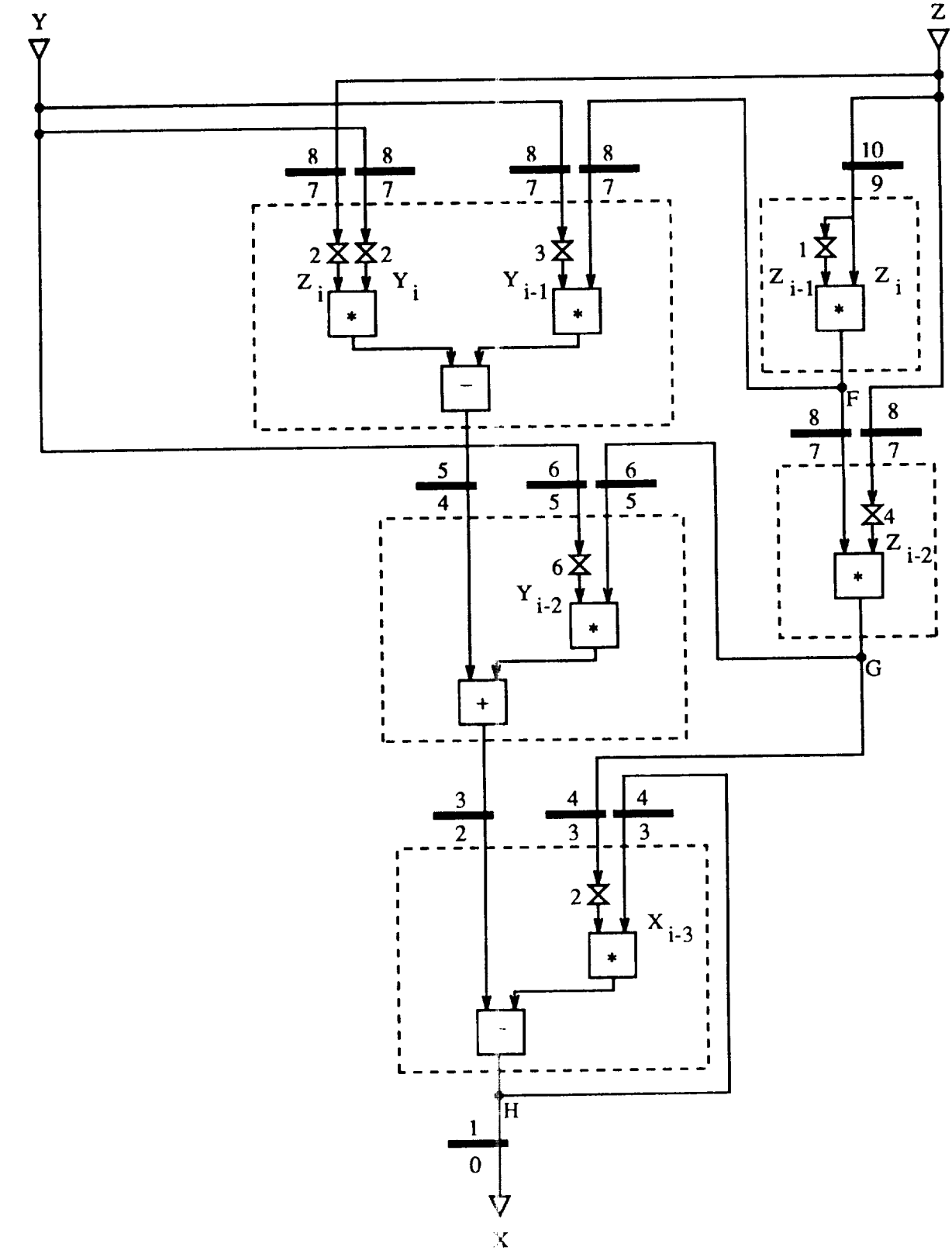

Figure 8. Restructured recurrence with pipeline delays

The next step is allocating ALS's following the four branch points, the Y and Z inputs and the two internal pipe branches. H is not included since it only feeds a single ALS. Figure 9 shows one allocation to these five ALS's. This step was very easy. Any time there is a branch, there are 6 ways to assign the labels a, b and c to the two or three resulting paths. It appeared that the four branches in this diagram could be set in any order and with any assignment without any backtracking being needed with subsequent branches. To test this hypothesis, the labels a, b and c were assigned to the branches without looking at the ALS labellings shown in Figure 2. A conflict occurred since both inputs to the doublet labelled 6 ended up being labelled a and none of the eight doublets satisfies this. Swapping the labels on branch F remedied this. Given the possibility of swapping ALU inputs, branch G was the only limiting factor in assigning the triplet to be ALS_3. ALS_0 is also possible but is considered more "valuable" since it is a possible output from the shift/delay units. Singlet ALS_{15} could be any other except ALS_{13}; singlet ALS_{14} could also be ALS_{15}; doublet ALS_6 could be ALS_4 or ALS_5, but these connect to the shift/delay units, and doublet ALS_7 could be any except ALS_{10} or ALS_{11}. A promising order of allocation, which would address the tightest limits first, would be to allocate ALS's with inputs from branches in the order of singlets before doublets before triplets and then by the number of labelled inputs. For this example at least, the cross constraints on ALS allocation due to the input labelling shown in Figure 2 presented negligible difficulty to the programming process.

This pipe will generate the vector X starting with X_3. It is necessary to initialise this pipeline by inserting the values X_0, X_1 and X_2 into ALS_7 and the F×F switch. These might be previously calculated by running the pipeline shown in Figure 6 and placing these values in the appropriate places and then shifting to this pipeline in a single time step. The actual NSC has this capability.

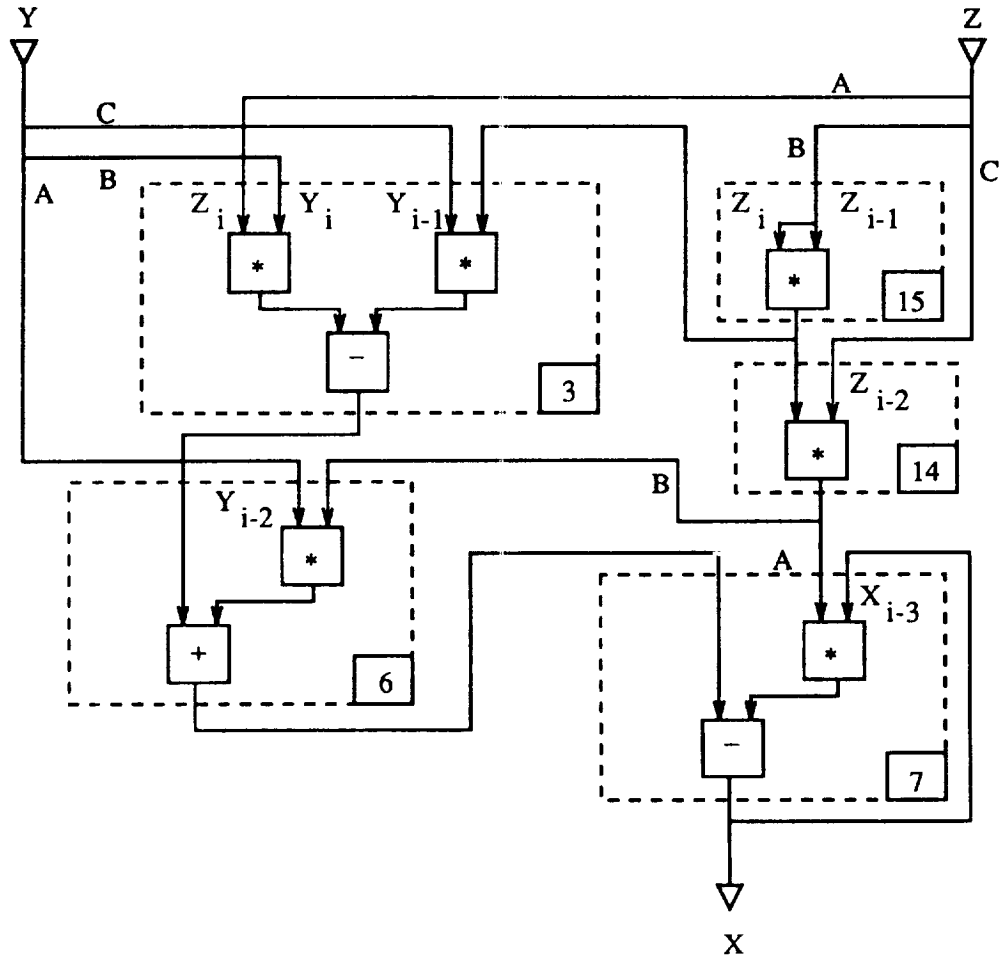

Figure 9. Restructured recurrence with pipeline delays

This method for calculating the recurrence uses 9 ALS's to generate values at the full 20 MFLOP speed for an effective utilisation of 22%.

Conclusions

Due to its design and purpose, the Navier-Stokes computer presents a difficult challenge to being used at or near full effectiveness. A visual editor with verification facilities has

been proposed as a support tool for the programmer in the seemingly necessary task of programming the NSC at the hardware level.

The study of two example applications suggests that the programming method presented is appropriate for the NSC's target: identical calculations performed on large arrays of data.

The study might also indicate useful modifications to the NSC design. In particular, the channels to memory need to be much more flexible and probably more numerous. In contrast, the extensive restrictions imposed by the use of permutation networks as opposed to full cross-bar switches, appear not to affect the programming task noticeably, and so may well remain or increase in the quest for more ALU's.

References

F. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

D. Nosenchuck, S. Krist and T. Zang, "On multigrid methods for the Navier-Stokes Computer", *3rd Copper Mountain Conference on Multigrid Methods*, Copper Mountain, Co., April 1987.

S. Tomboulian, T. Crockett and D. Middleton, "A visual programming environment for the Navier-Stokes Computer", *Seventeenth International Conference on Parallel Processing*, St. Charles, Ill., August 1988.

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181691		

4. Title and Subtitle	5. Report Date
PROGRAMMING THE NAVIER-STOKES COMPUTER: AN ABSTRACT MACHINE MODEL AND A VISUAL EDITOR	August 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David Middleton, Tom Crockett, and Sherry Tomboulian	Interim Report 5
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

The Navier-Stokes Computer is a parallel computer designed to solve Computational Fluid Dynamics problems. Each processor contains several floating point units which can be configured under program control to implement a vector pipeline with several inputs and outputs. Since the development of an effective compiler for this computer appears to be very difficult, machine level programming seems necessary and we have studied support tools for this process. These support tools are organised into a graphical program editor.

A programming process is described by which appropriate computations may be efficiently implemented on the Navier-Stokes computer. The graphical editor would support this programming process, verifying various programmer choices for correctness and deducing values such as pipeline delays and network configurations. Step by step details are provided and demonstrated with two example programs.

We also describe an abstract reconfigurable vector processor in place of the actual processor used in the Navier-Stokes computer. This abstract node provides a firm and simple definition around which the editor and the programming activity can be designed. It is a subset of the actual node, displaying the important facilities of an actual node while hiding various implementation details from the programmer. Because of the intended applications, aspects of multiprogramming such as synchronisation and communication have been ignored; it is expected that simple program barriers will suffice. Designing the abstract node to match the machine facilities used during the programming process provides some early feedback on features provided in the actual node processors.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
programming vector computers, reconfigurable computer, multiple-function-unit computer	61 - Computer Programming and Software Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	30	A03