NASA Contractor Report 181615

ICASE REPORT NO. 88-6

ICASE

A VISUAL PROGRAMMING ENVIRONMENT FOR THE NAVIER-STOKES COMPUTER

(NASA-CR-181615) A VISUAL PROGRAMMING ENVIRONMENT FOR THE NAVIER-STOKES COMPUTER Final Report (NASA) 19 p CSCL 09B

N88-18299

Unclas
G3/61 0128064

Sherryl Tomboulian

Thomas W. Crockett

David Middleton

Contract No. NAS1-18107
January 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

A Visual Programming Environment for the Navier-Stokes Computer

Sherryl Tomboulian Thomas W. Crockett
David Middleton

Institute for Computer Applications in Science and Engineering

ABSTRACT

The Navier-Stokes Computer is a high-performance, reconfigurable, pipelined machine designed to solve large computational fluid dynamics problems. Due to the complexity of the architecture, development of effective high-level language compilers for the system appears to be a very difficult task. Consequently, a visual programming methodology has been developed which allows users to program the system at an architectural level by constructing diagrams of the pipeline configuration. These schematic program representations can then be checked for validity and automatically translated into machine code. The visual environment is illustrated by using a prototype graphical editor to program an example problem.

This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the authors were in residence at ICASE.

Authors' addresses: ICASE, M.S. 132C, NASA Langley Research Center, Hampton, VA 23665. Electronic mail: sjt@icase.arpa, tom@icase.arpa, koala@icase.arpa.

A Visual Programming Environment for the Navier-Stokes Computer

Sherryl Tomboulian Thomas W. Crockett
David Middleton

Institute for Computer Applications in Science and Engineering

1 Introduction

The Navier-Stokes Computer (NSC) [6], developed at Princeton University with funding from NASA, is a special-purpose, high-performance parallel system designed for very large computational fluid dynamics (CFD) applications. The architecture consists of multiple processing nodes arranged in a hypercube configuration. Each node contains a few dozen functional units which can be reconfigured dynamically into one or more vector pipelines.

The architecture has some features resembling those of Multiflow's TRACE[1] computers [4] and CDC's CYBERPLUS[2] [1,3], such as multiple function units and long instruction words. However, there are significant differences which appear to make development of effective high-level language compilers a very difficult problem. As an alternative, a visual programming methodology is presented which employs a graphical interface to assist the user in programmming the NSC at the machine architecure level.

[1] *Multiflow* and *TRACE* are trademarks of Multiflow Computer, Inc.
[2] *CYBERPLUS* is a trademark of Control Data Corporation.

A brief overview of the NSC is given first, and some of the difficulties in programming it with conventional methods are discussed. The design for a visual programming environment is then described, and a prototype version is used to illustrate the concepts for a sample problem. Conclusions based on experience with the prototype system are reported.

2 NSC Architecture

The major architectural components of the Navier-Stokes Computer are described here. The focus is on the individual nodes, rather than on the system as a whole, since it is the internal design of the nodes which makes the NSC a novel architecture. The information presented is a considerable simplification of the actual design, with many details omitted for the sake of clarity. The description given is sufficient for an understanding of the visual environment described in this paper. The final design of the NSC hardware is not complete at this writing, so some adjustments to the following may be needed in the future.

Each node contains 32 functional units. Every functional unit can perform floating-point operations, and some of them can also perform either integer/logical operations or max/min computations. In addition, each functional unit has an associated register file which can be used to store constants or intermediate values, as well as to buffer data to adjust for pipeline timing delays. The functional units are hardwired into three types of *arithmetic-logic structures* (*ALSs*), called *singlets*, *doublets*, and *triplets*, which contain respectively 1, 2, or 3 floating-point units.

Memory is arranged in 16 planes of 128 Mbytes each, for a total memory of 2 Gbytes per node. In addition, there are 16 double-buffered data caches. Two *shift/delay units* are provided to aid in reformatting memory data into multiple vector streams. A complex programmable switching network routes data among ALSs, memory planes, caches, and shift-delay units. Communication between nodes is handled by means of a *hyperspace router*. The various hardware components are configured into vector pipelines during execution by programming the switches. Multiple pipelines may be set up to run in parallel. The pipeline configurations may be rapidly modified under program control as the computation proceeds through different phases. Scalars are treated as vectors of length one. A simplified diagram of the data path architecture is shown in Figure 1.

Control flow is even more complex. A central sequencer provides high-level control flow, but independent DMA controllers associated with each memory and cache plane pump data through the pipelines. An elaborate interrupt scheme is used to signal pipeline completions, evaluate conditional expressions, and trap exceptions.

Projected peak performance of the system is quite high, with a maximum rate of 640 MFLOPS per

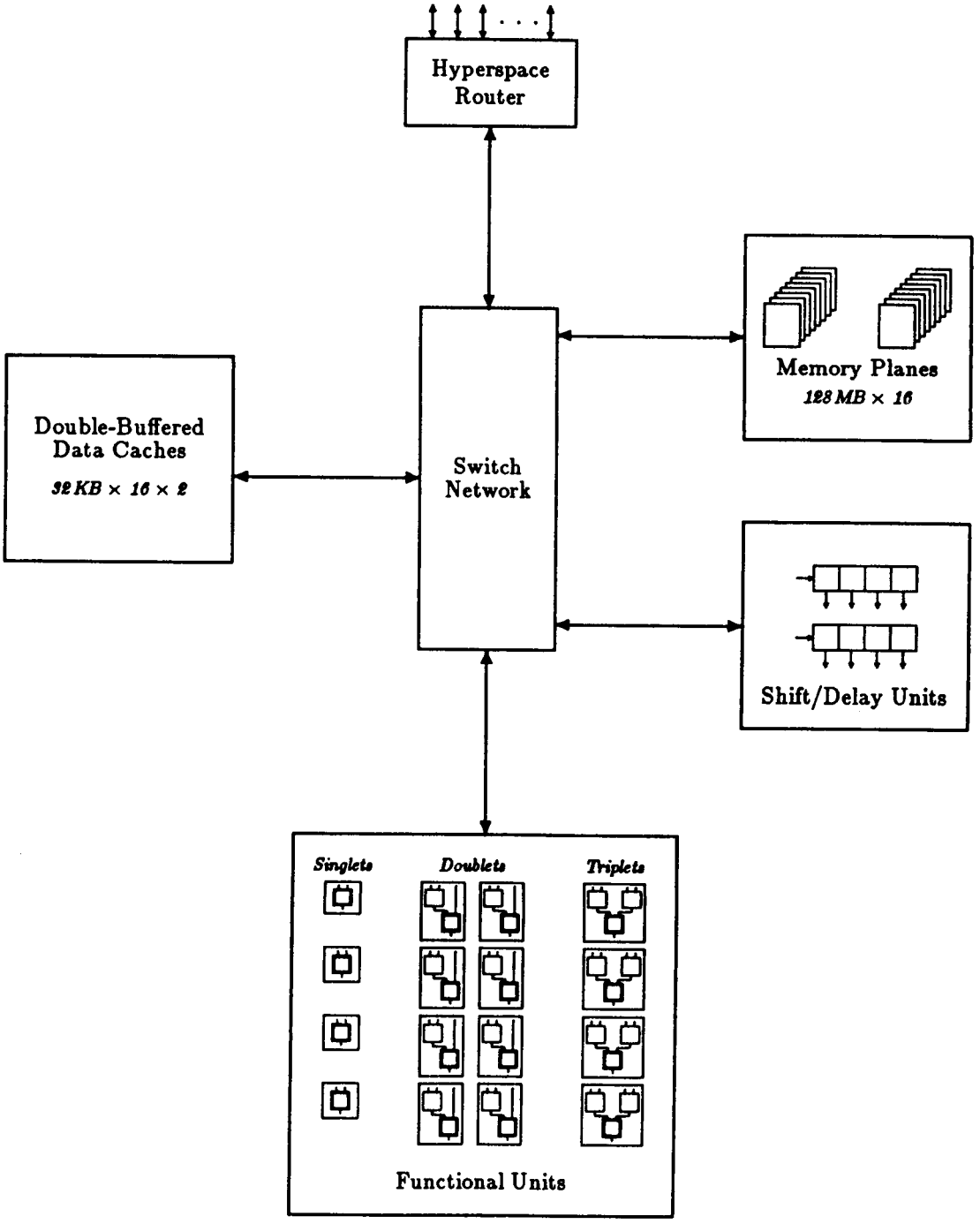


Figure 1: Simplified diagram of the datapath architecture of the Navier-Stokes Computer.

node. A 64-node NSC would have a total memory of 128 Gbytes and maximum performance of 40 GFLOPS [6].

3 Programming Considerations

There are several features of the NSC architecture which make compilation of high-level languages into efficient object code a difficult task. One of the most serious is the organization of memory into separate planes. During an instruction (vector operation), a function unit can read or write in only a single memory plane, and multiple function units working in the same memory plane can cause contention problems. This causes serious problems for a compiler in trying to decide where to allocate variables, since the optimum layout for one pipeline may be unworkable for the next. In some cases, it may be necessary to maintain multiple copies of arrays, or to relocate them between phases of the computation. Another problem arises since the function units within each ALS are not constructed identically. Only a single unit can perform integer operations, and another unit has circuitry for min/max computations. This, coupled with the distinctions between singlets, doublets, and triplets, complicates the problem of mapping function units onto expression graphs. Generation of control code is made more difficult by the presence of multiple sites of control (sequencer, DMA units, interrupts, etc.) which must be carefully orchestrated to insure that all possible actions are mutually consistent. Numerous other details tend to complicate programming as well. Any of the individual problems could probably be handled successfully, but they tend to interact with each other, making the overall problem more difficult than the sum of the subproblems. Given current compiler technology, it is difficult to see how all of these considerations can be handled simultaneously while still producing code that can achieve high utilisation of 32 function units. It has been estimated that construction of a FORTRAN compiler for the NSC would take about three years, and the performance of the resulting code relative to other high-performance computers is in doubt.

Because of these problems, it seems that a programming methodology more closely tied to the architecture might deliver better performance. Traditionally, this has been accomplished by writing assembly language programs for performance-critical applications. Unfortunately, the NSC lacks anything resembling a conventional assembly language. Each instruction must be specified in a complex hierarchical microcode which contains specific control for every function unit, register file, switch setting, DMA unit, etc. The effect of an instruction is to completely specify the pipeline configuration and function unit operations for the entire machine. This requires a few thousand bits of information per instruction, encoded in dozens of separate fields. Therefore, hand-written microprograms are clearly not practical for the NSC.

4 A Visual Programming Environment

In an effort to simplify programming at the architectural level, a visual programming methodology has been developed. This approach is based on an informal manual technique which evolved among applications researchers at Princeton University and NASA's Langley Research Center. Using this technique, programs were designed by hand-drawing a series of pipeline configurations, each representing one stage, or loop body, within the overall program. The diagram in Figure 2, which has been cleaned up for presentation purposes, is a typical example of this style. The figure represents a point Jacobi update for the 3-D Poisson equation on a uniform grid with a residual convergence check [6]. The equation for the update is given by

$$\begin{aligned} u_{ijk}^{(m+1)} &= u_{ijk}^{(m)} + \frac{h^2}{6} R_{ijk}^{(m)} \\ R_{ijk}^{(m)} &= \frac{1}{h^2}(u_{i+1,jk}^{(m)} + u_{i-1,jk}^{(m)} + u_{i,j+1,k}^{(m)} + u_{i,j-1,k}^{(m)} + u_{ij,k+1}^{(m)} + u_{ij,k-1}^{(m)} - 6u_{ijk}^{(m)}) - G_{ijk} \end{aligned} \tag{1}$$

The natural evolution of this manual technique suggested that an automated environment in which pipeline instructions were drawn interactively on a graphics display and then automatically translated to microcode could be an effective way of programming the NSC at the machine level.

The concept of visual programming is not new, but it has become increasingly practical as workstations with high resolution graphics have become widely available [5]. A recent survey of visual programming techniques can be found in [2]. This method seems to be a natural approach for programming data flow and pipelined architectures. Visual programming techniques have been applied to parallel architectures before, but for different architectural models. A prominent example is Poker [7], which is a parallel programming environment designed to support the CHiP computer.

The scope of this project has purposely been limited to internal programming of individual nodes, since this area is the source of greatest difficulty. If needed, techniques similar to those used in Poker could be applied to the larger multi-node environment. Although the design has been tailored specifically to the NSC, the same general approach could be used for other reconfigurable pipeline machines.

Three major goals were established for the NSC visual programming environment. The first is that the representation have a one-to-one correspondence with the functional model of the machine, so that everything could be specified precisely if necessary. However, an effort would be made to choose appropriate defaults wherever possible in order to minimize the amount of detail required. The defaults could be easily overridden when required. The second goal is that the graphical representation be easy to program and clearly represent the semantics so that a programmer looking at an instruction would immediately see the intent. The third is that the environment would do all it could to ease the user's task by preventing or indicating syntactic errors and violations of hardware constraints as the program

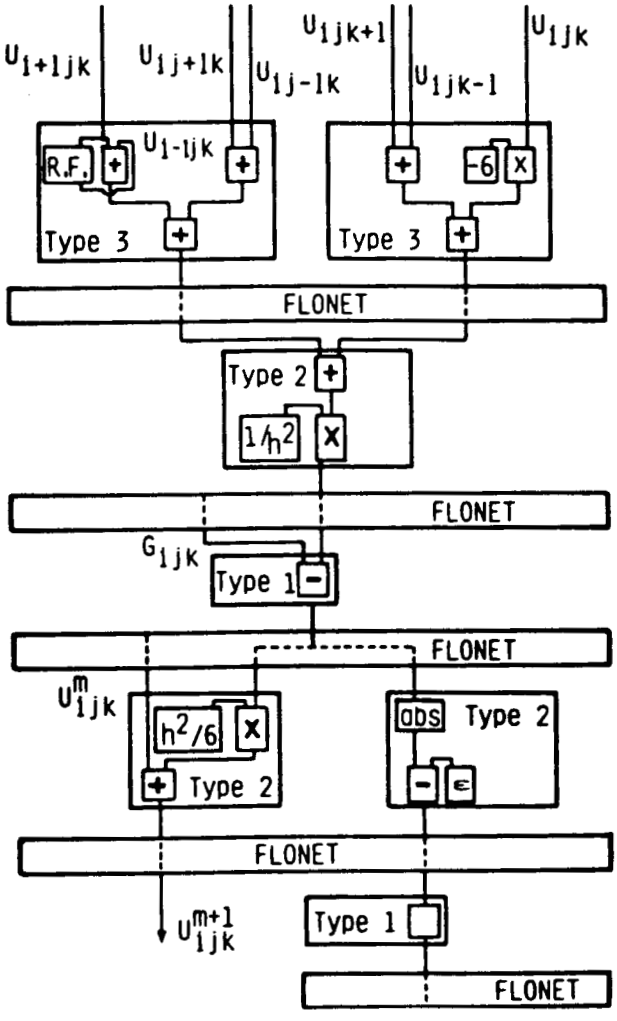


Figure 2: Example pipeline diagram generated as an aid to program design. (From [6].) **FLONET** refers to a portion of the switch network.

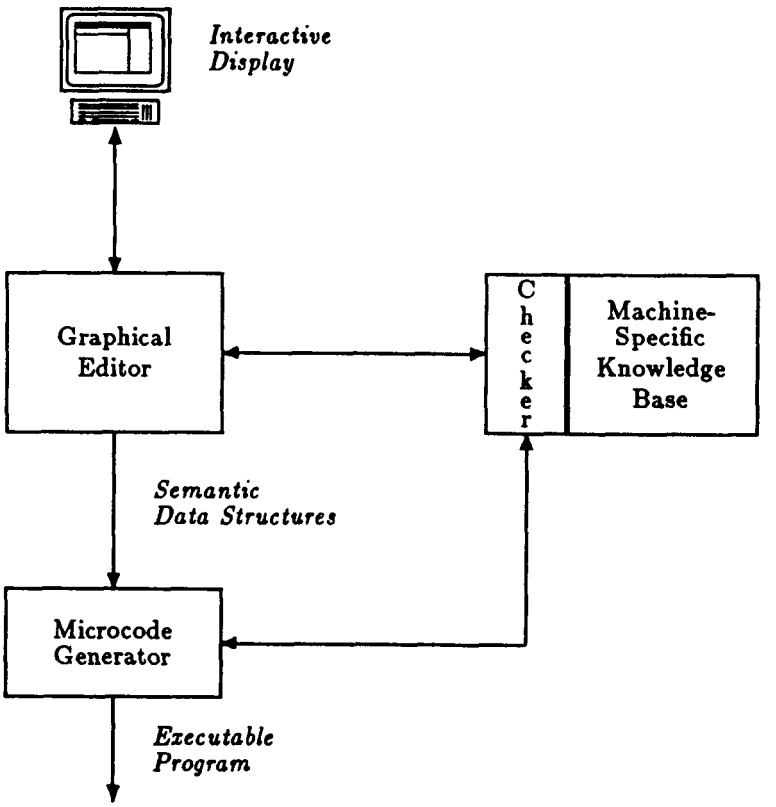


Figure 3: Major components of the visual programming system.

is entered. This error-checking philosophy is similar to that embodied by syntax-directed editors for textual programming languages. More extensive checking could be done when the visual representations are translated to microcode, and any additional errors would be visually presented to the programmer.

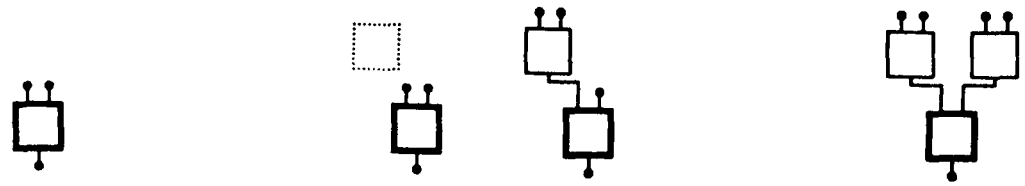
The design for the visual environment contains three major components, a graphical editor, a *checker*, and a microcode generator. Figure 3 shows the interaction between these components and the user. The graphical editor provides the usual operations found in an editor, such as the ability to enter new input, modify or delete existing data, and save the results. However in this case, the objects being operated on are graphical rather than textual. The graphical editor also is responsible for extracting information from the pictures and storing it in internal data structures. Two types of internal data are distinguished. One type consists of information which is needed solely to manage the graphical display, such as the position of images on the screen. The other type consists of semantic information which

is needed in order to generate microcode. Since the semantics are represented graphically, both types of information are needed in order to reconstruct the display. But in order to generate code, only the semantic information is needed.

The checker contains, in a knowledge base or other suitable representation, detailed information about the architecture of the NSC, so far as it is relevant to the programming process. This includes various machine parameters such as the number and types of function units, their organization into ALSs, the number and size of memory planes, etc. More importantly, the checker also knows all of the rules about conflicts, constraints, asymmetries and other restrictions in the NSC architecture. The graphical editor calls on the checker at appropriate points during interaction with the user to validate the information being input. Any errors are flagged as soon as they are detected. In addition, the graphical editor uses the checker's knowledge of the architecture to reduce the possibilities for making errors. For example, if the user has routed the output from one function unit to a particular memory plane, the graphical editor will not let him send the output of a second unit to the same plane. The philosphy is similar to that embodied by syntax-directed text editors, with the goal being to assist the user in developing correct programs despite the complexity and numerous restrictions of the architecture. Another advantage of having a checker is that it helps to make the whole visual environment more robust in the face of changes to the machine design. Some changes can be handled merely by updating the knowledge base, with minimal impact on the graphical editor and microcode generator.

Once a complete program (or consistent program fragment) has been defined, the microcode generator uses the semantic data structures created by the graphical editor to generate machine code for the NSC. The checker is invoked again at this point to perform a thorough check of global constraints and other conditions which may not be practical to check during the editing process.

In order to test the concept of visual programming for the NSC, a prototype graphical editor/assembler was designed and implemented. The prototype focusses mainly on the graphical editor portion of the design, in order to determine whether the great level of detail needed in the microcode instructions can be conveniently presented pictorially, and to assess the ease of programming with this type of interface. The checker is not present as a separate entity, although some checking functions are incorporated into the graphical interface. Since the final design of the NSC is not complete, and there is no means of running actual NSC programs, the prototype produces only the semantic data structures as output, rather than the actual microcode instructions. The semantic data can be thought of as a pseudo-code representation of the instructions.



Singlet Doublets Triplet

Figure 4: ALS icons.

5 Programming Example

The major features of the visual environment are illustrated here using the prototype graphical editor to program the example problem from the previous section. The central concept is that visual objects, or *icons*, are used to represent architectural components of the NSC at a suitable level of abstraction. The user manipulates these icons interactively to construct a program. Subimages within each icon are also meaningful, providing the interface to an additional level of program detail. A high-resolution bit-mapped display is used as the drawing surface. Interaction is provided primarily with a "mouse", augmented with a keyboard for some operations.

In the prototype, icons consist principally of the three different ALS types (Figure 4). Two representations of the doublet are provided, since doublets may be configured to operate as singlets by bypassing one of the functional units. Functional units are shown as squares within the icon, with the "double box" units having integer/logical as well as floating-point capabilities. Other icons which would be useful, but are not currently implemented, include memory planes and shift-delay units.

In addition to the icons, a variety of other visual devices are employed. These include pop-up menus and subwindows, "buttons", "sliders", and even text fields, where appropriate. The prototype is implemented on a Sun-3 workstation using SunView[3] graphics software.

Figure 5 shows the basic display window used. The right hand side is a "control panel" area used to select icons and specify various editor operations. The large area in the center is the drawing space in which pipeline diagrams are constructed. Informational and error messages are displayed in the narrow strip across the top. The region at the left is reserved for control flow specifications and variable declarations, which are not implemented in the prototype.

[3] *Sun-3* and *SunView* are trademarks of Sun Microsystems, Inc.

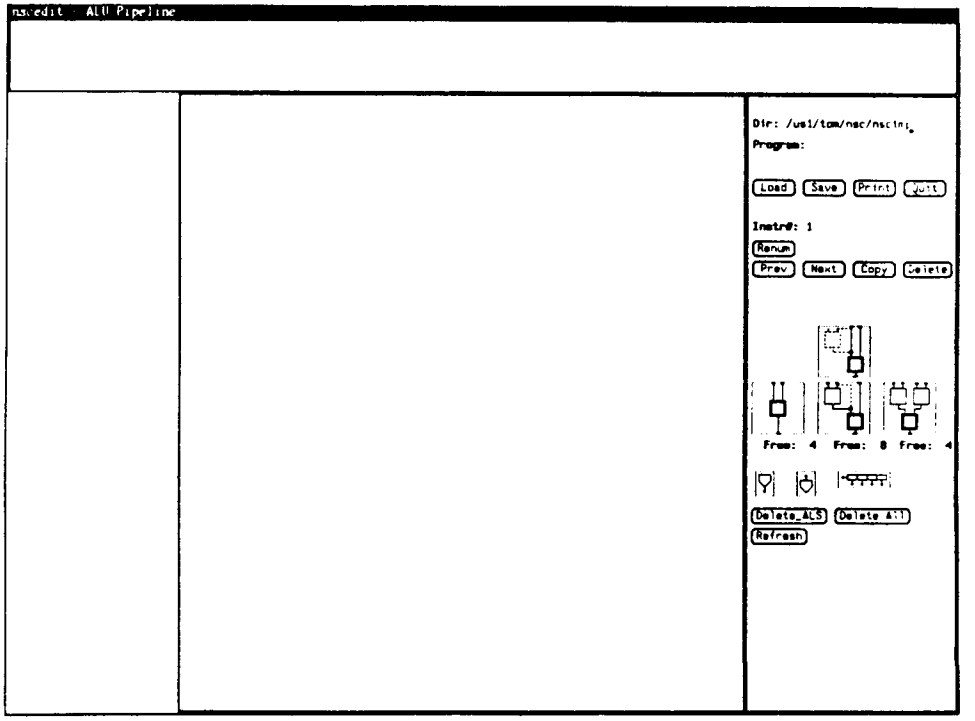


Figure 5: Display window for the visual environment.

To construct a program, a user defines a series of pipeline diagrams. Each pipeline corresponds to a single instruction, or one line of code, in a more conventional language. Control panel operations provide the usual editor operations to insert, delete, copy, and renumber pipelines, as well as to scroll forward or backward or jump to a specific pipeline.

The first step in constructing a pipeline diagram is to select the needed ALSs and position them in the drawing area of the screen (Figure 6.) This is accomplished by moving the mouse pointer into the control panel area and selecting the appropriate icon, then "dragging" the outline of the ALS to its desired location. The process is repeated until all of the needed ALSs have been selected (Figure 7).

The next step is to specify the inputs and outputs of the function units. These are selected by "mousing" on the I/O pads (short wires terminated by small black circles). A menu pops up showing the available choices. These may be either external connections to other function units, caches, memories, or shift/delay units, or else internal connections for feedback loops or register file data. Timing delays, needed for proper alignment of vector streams, may be introduced by routing input data into a circular queue in a register file and then retrieving the value a number of clock cycles later when it appears at the head of the queue.

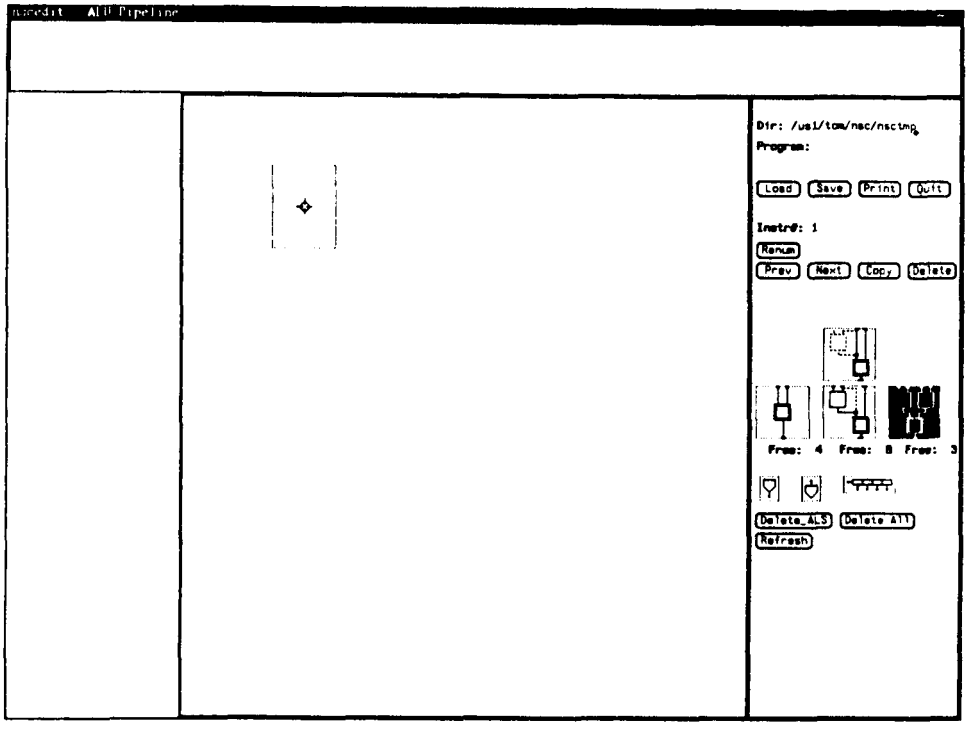


Figure 6: Selecting and positioning an icon.

Figure 8 illustrates the process of connecting the output from one function unit to the input of another. The mouse controls a "rubber-band" line which conceptually indicates a wiring connection between the two pads. The checker is used during this operation to ensure that only legal connections are attempted. The microcode generator would later derive switch settings by interrogating the connection tables built by the graphical editor.

In the case of a cache or memory connection, additional information is needed to program the DMA units. This is handled by a pop-up subwindow, in which the cache or memory plane number, variable name or starting address, stride, etc. are specified.[4] (See Figure 9.)

Note that the use of pop-up menus and windows is crucial to our approach. By hiding ancillary information until it is needed, the amount of detail displayed in the pipeline diagrams is reduced to a manageable level. Menus and subwindow templates also serve to prompt the user for needed information and remind him of his choices, both valuable services in an environment as complex as the NSC architecture.

The third and final step is to program the functional units by specifying the arithmetic or logical

[4]Several improvements in the graphical treatment of memories and caches have recently been designed, but have not yet been fully integrated into the prototype system.

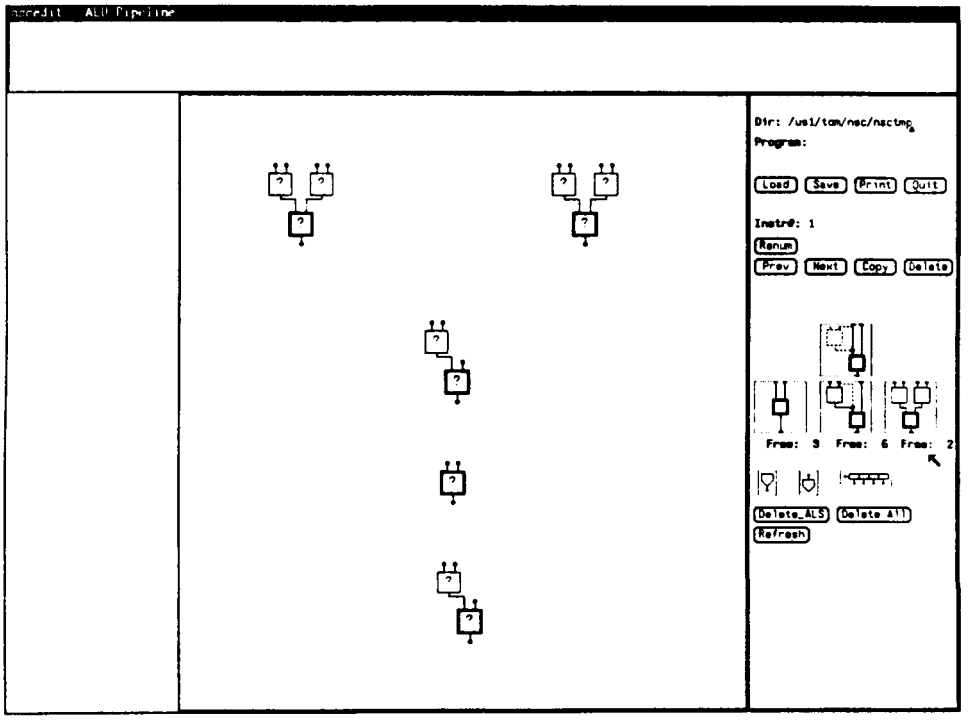


Figure 7: Display after all ALSs have been positioned.

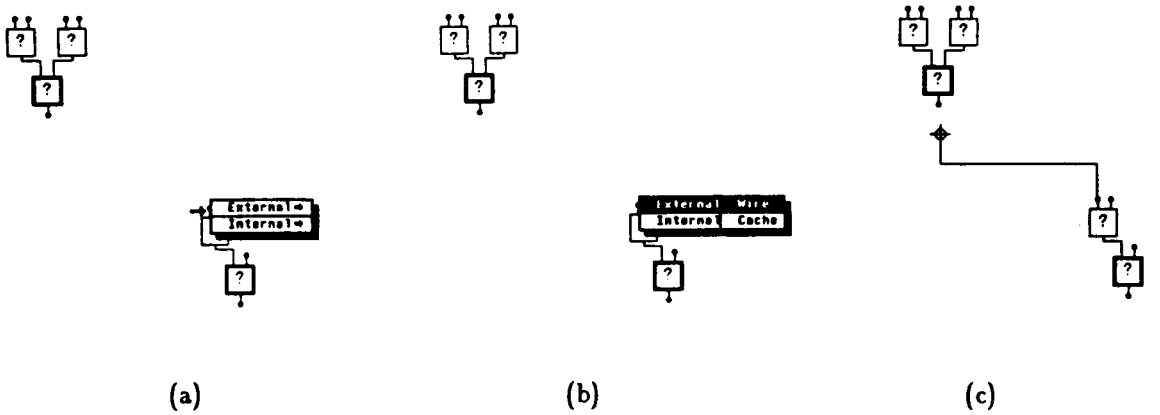


Figure 8: Establishing connections between function units.

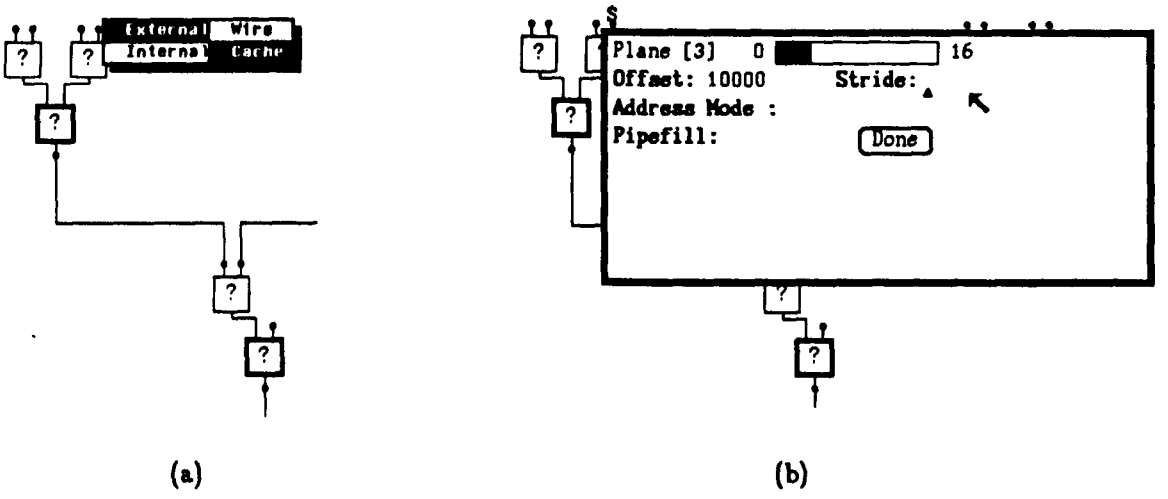


Figure 9: Pop-up subwindow for specifying cache connections.

operations which they are to perform. Once again this is done with a pop-up menu (Figure 10). The menu appears when the mouse is used to select a function unit within an ALS. Figure 11 shows the completed pipeline diagram for the point Jacobi iteration of Equation 1.

6 Conclusions

While the results based on the prototype graphical editor should be regarded as preliminary, it appears feasible to implement a complete visual programming environment for the Navier-Stokes Computer. This environment would clearly be more convenient and faster to use than hand-written microcode. The improvements derive from several factors. First, the visual representation more clearly reflects the hardware architecture and program intent than do reams of textual microassembler code. The data-flow style of the diagrams also seems to be a natural way for humans to describe computations. In addition, information hiding with subwindows can be used to effectively reduce the amount of low-level detail which must be displayed and assimilated at one time. This is somewhat analogous to the use of macros and subroutines in textual languages. Another advantage is that the detailed knowledge of architectural intricacies built into the visual environment reduces the possibility of writing erroneous programs and errors are caught sooner when they do occur.

On the other hand, programming at this level, even with the graphical interface, is a tedious process. The amount of machine-level detail which must be specified requires that the programmer have a good understanding of the hardware design. The user must focus not only on solving his problem, but also on

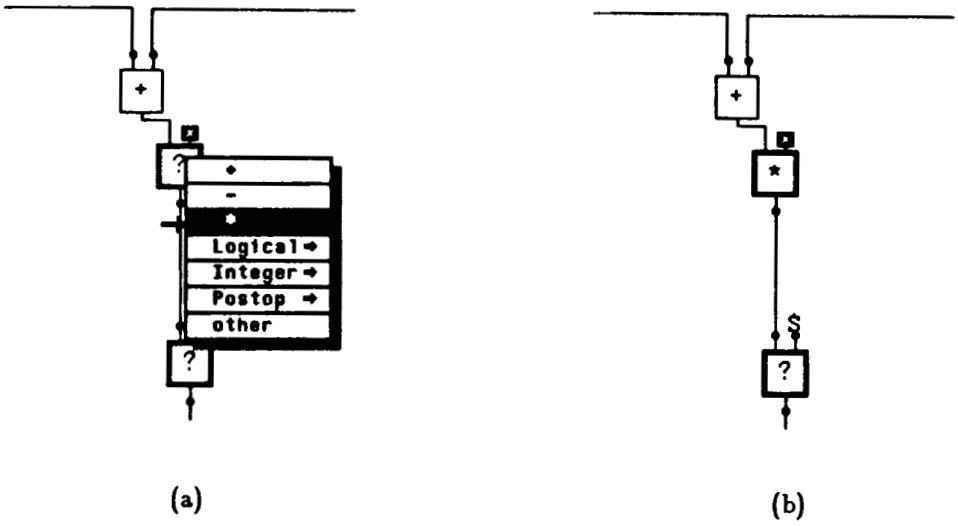


Figure 10: Programming individual function units.

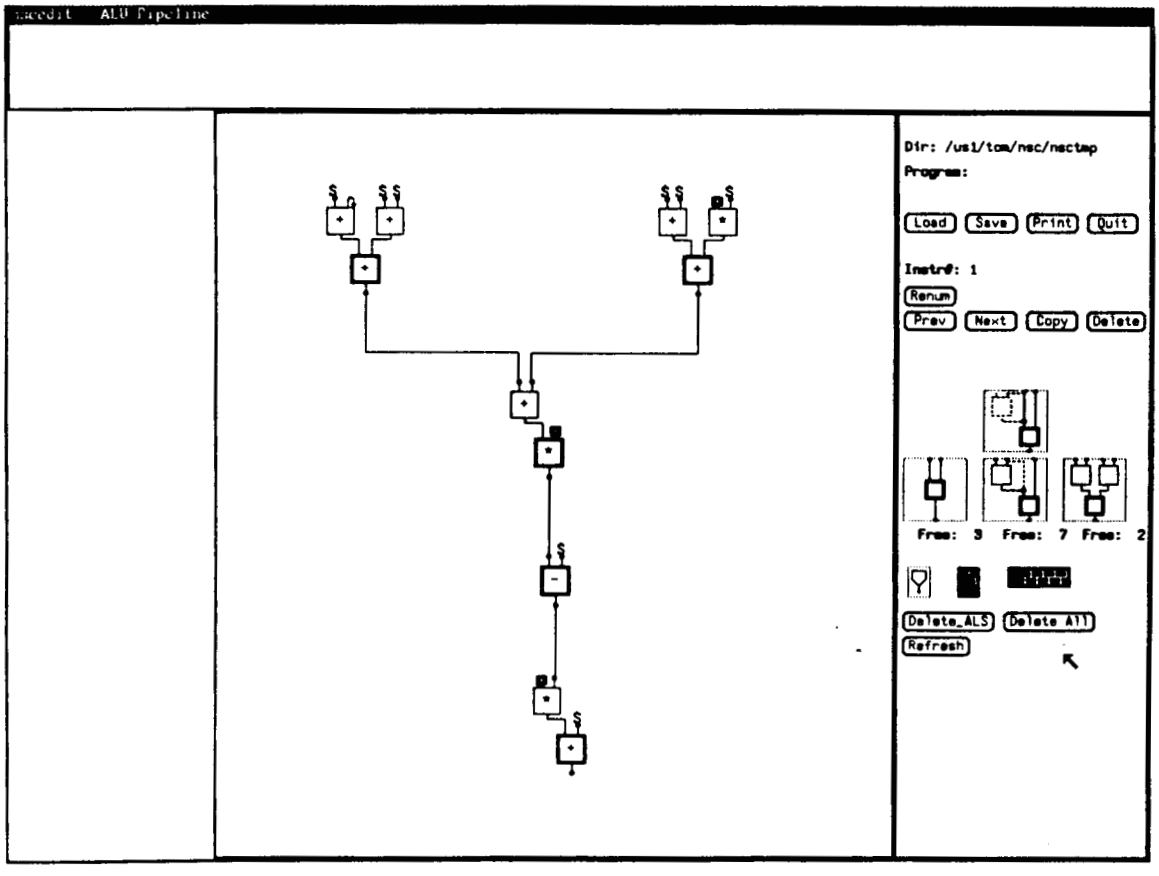


Figure 11: Completed pipeline diagram for the point Jacobi iteration.

mapping his problem onto this very complex architecture. So given a choice, a higher-level programming environment would be preferable.

One approach to reducing the complexity is to use a simpler architectural model, perhaps a subset of the NSC. The tradeoff here is between performance and programmability. By ignoring certain features of the architecture, it may become easier to program, but performance may be adversely affected in some situations. Initial examination of this approach has shown that some abstraction is possible, but the performance ramifications are unclear.

Experience with the prototype visual environment also indicates that implementation of a complete, production-quality system would be a significant software development project, resulting in at least a few tens of thousands of lines of code. Most of this would be devoted to the graphical interface and the checker, with smaller portions devoted to code generation and traditional editor functions.

The visual environment could potentially be extended to include debugging features. During execution, each new instruction would display the corresponding pipeline diagram, annotated to show data values flowing through the pipeline. This could help to pinpoint timing errors, as well as other bugs in the program. The visual environment might also be useful as a back end to a compiler, displaying the results of the compilation process.

In summary, a visual programmming environment offers several advantages for efficiently programming a reconfigurable pipeline architecture such as the NSC. However, it is still essentially a low-level programming language, and requires a significant implementation effort in order to become a useful tool. It remains to be seen whether this approach can compete with compiled high-level languages over the long term.

References

[1] R.G. Babb, L. Storc, W.C. Ragsdale, "A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS", *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 845-848.

[2] S. Chang, "Visual Languages: A Tutorial and Survey", *IEEE Software*, Vol. 4, No. 1, Jan. 1987, pp. 29-39.

[3] Control Data Corporation, "CYBERPLUS Hardware Reference Manual", Publication No. 77960981.

[4] J.A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", *Computer*, Vol. 17, No.7, July 1984, pp.45-53.

[5] R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming", *Computer*, Vol. 18, No.8, Aug. 1985, pp.51-59.

[6] D.M. Nosenchuck, S.E. Krist, T.A. Zang, "On Multigrid Methods for the Navier-Stokes Computer", in *Multigrid Methods*, S. McCormick and K. Stuben (eds.), Marcel-Dekker, 1988.

[7] L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer*, Vol. 17, No.7, July 1984, pp. 27-36.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181615 ICASE Report No. 88-6		

4. Title and Subtitle

A VISUAL PROGRAMMING ENVIRONMENT FOR THE NAVIER-STOKES COMPUTER

5. Report Date

January 1988

6. Performing Organization Code

7. Author(s)

Sherryl Tomboulian, Thomas W. Crockett, and David Middleton

8. Performing Organization Report No.

88-6

10. Work Unit No.

505-90-21-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18107

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

Submitted to 1988 International Conference on Parallel Processing

16. Abstract

The Navier-Stokes computer is a high-performance, reconfigurable, pipelined machine designed to solve large computational fluid dynamics problems. Due to the complexity of the architecture, development of effective high-level language compilers for the system appears to be a very difficult task. Consequently, a visual programming methodology has been developed which allows users to program the system at an architectural level by constructing diagrams of the pipeline configuration. These schematic program representations can then be checked for validity and automatically translated into machine code. The visual environment is illustrated by using a prototype graphical editor to program an example problem.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
visual programming, parallel environment	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	18	A02