The Warp Machine on Navlab

Jill D. Crisman, *Member, IEEE,* and Jon A. Webb

***Abstract*—We review the history of the Carnegie-Mellon Warp machine on Navlab, an autonomous land vehicle, and describe three Navlab vision systems implemented on the Warp machine. We then critically evaluate components of Warp in light of this experience. The Warp machine was used to implement stereo vision for obstacle avoidance and color-based road-following systems. The stereo-vision system was FIDO, which is descended from some of the earliest work in vision-guided robot vehicle navigation. Two color-based road following systems were implemented; one adapted conventional vision techniques to the problem of road recognition, and the other used a neural network-based technique to "learn" road following on-line. Finally, we conclude with observations on the utility of Warp on Navlab, the value of applications integration with machine development, the limitations of the "attached processor" model, and recommendations for future systems.**

***Index Terms*—Autonomous navigation, mobile robots, neural networks, road following, stereo vision, supercomputers, warp computer.**

I. Introduction

THE Carnegie-Mellon Warp machine is a systolic array computer developed by H. T. Kung's group and used for many applications including signal and image processing and mobile robot control [1]. We relate the history of the use of the Warp machine on Navlab (Navigation Laboratory) and evaluate the Warp machine in light of this experience. As we will demonstrate, the Warp and Navlab projects influenced each other in several ways; this influence led to increased capabilities in the Warp machine and useful applications experience, as well as increased capabilities for Navlab.

We begin with a short history of the Warp machine on Navlab. Next we describe the major Navlab systems that were implemented using the Warp machine. Then we evaluate the Warp machine using experience from these systems.

Manuscript received November 15, 1989; revised December 10, 1990. Research on Navlab (including Parallel Vision research) was supported by the Defense Advanced Research Projects Agency, DOD, (DARPA) under Contracts DACA76-86-C-0019, DACA76-85-C-0003, and DACA76-85-C-0002, all monitored by the Engineer Topographic Laboratories. The Warp project was supported in part by DARPA under Contract N00039-85-C-0134, monitored by the Space and Naval Warfare Systems Command, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533. The ALVINN work was supported by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533, by the National Science Foundation under Grant EET-8716324, and by DARPA monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251. A version of this paper appeared in *Vision and Navigation: The Carnegie Mellon Navlab,* C. Thorpe, Ed. (Kluwer Academic Publishers, 1990).

J. D. Crisman was with the School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. She is now with the Electrical and Computer Engineering Department, Northeastern University, Boston, MA 02115.

J. A. Webb is with the School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9142750.

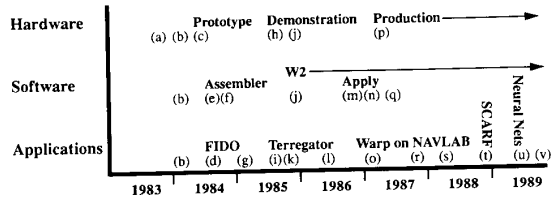


Fig. 1. History of Warp.

II. History of the Warp Machine on Navlab

This section traces the history of the development of the machine, its software, and its application on Navlab, and discusses the motivations that led to key decisions. Fig. 1 is a timeline of the key developments referred to here in parentheses.

Warp's origins were in two-level pipelined systolic arrays described by Kung *et al.* [5], [8] in the early 1980's. The systolic array formed one pipeline because the linear array of cells could pipeline data from one cell to the next, and within each cell the floating-point pipeline formed another. These designs were shown to be capable of convolution.

The introduction of the Weitek 1032 floating point chips made it possible to implement a powerful machine based on these ideas using ordinary engineering effort, i.e., without custom VLSI and with a moderate number of processors. A machine using these chips was designed (a), and it was shown to be capable of performing one- and two-dimensional convolution as well as the fast Fourier transform (FFT) [6].

The first Warp cell included the Weitek floating point chips, which were fed data from a pipelined register file, two input and output queues connecting each cell, and some on-board cell memory [7]. Addresses were supplied externally via a third queue. No data-dependent branching or address generation was possible.

A group of researchers, including hardware, software, and applications designers, began planning the design of the Warp computer (b). The design of the machine changed rapidly and became much more general. Data-dependent control flow and program memory was added. A crossbar, originally with limited interconnection and later with full generality, was added to connect the various functional units on the cell. The pipelined register files were replaced by random-access register files.

The Warp cell still had several features that were eliminated later. Addressing was factored out of the cells; one board generated addresses for the array, so that cells could not generate data-dependent addresses. Cell-to-cell communication included features that were rarely used.

0162-8828/91/0500-0451$01.00 © 1991 IEEE

The Warp cell was built in prototype form, as a two-cell array with an interface unit, called the "demonstration" machine (c). Assemblers were developed (e and f). Even before assemblers or simulators were available, programs such as affine transformation, clipping, histogram, median filtering, and binary image processing were being written (d).

Carnegie-Mellon selected General Electric and Honeywell as industrial partners at the beginning of 1985. They participated in the construction of the first two-cell demonstration machine (h). The array consisted of two Warp cells and an interface unit. It was controlled and fed data by a Sun 2, which also ran an applications code not running on the Warp array. The external host, consisting of MC68020 processors (called the "cluster" processors) and associated memories especially for feeding data to and from the Warp cell array, was still being designed and constructed in cooperation with the industrial partners.

FIDO, a stereo vision system used to drive a robot vehicle, was a key application of the Warp machine in the early part of the project (d). It was proposed to speed up FIDO by a factor of ten, from about 30 s/step to about 3 s/step.

The start of the Parallel Vision and Road Following projects (g) led to early use of the Warp machine in real situations. In most hardware projects, applications of the hardware to real problems occurs only after much of the software and hardware is already developed. But these projects helped provide focus and direction for the Warp project even as the hardware and software were being defined. A simple color-based road-following program was implemented on Warp (i), and used to drive the Terregator in the fall. To our knowledge, this is the first application of a supercomputer to actual control of a robot vehicle. These runs set records for speed and distance (up to several hundred meters at 0.5 km/h) of the Terregator.

The W2 compiler was developed in parallel with the applications of the demonstration Warp array. An important change was made that required construction of a second demonstration machine to make code generation easier and to reduce code size [1, Section IVa]. Shortly after the hardware change was completed (j) the old machine (and assembler programming) was retired and the new machine (with W2) was used exclusively.

Navlab was intended to be self-contained; we could not control it remotely, as we did with the Terregator. Installing Warp on Navlab (j) led us to consider issues of cooling of the Warp cells and vibration. As we later learned, the critical issues were cooling of the cluster processors and memory boards and connector damage as the Warp cells were removed and replaced in the backplane. The Warp cells were not difficult to cool because they dissipated much less heat per area than the commercial external host boards, which were more tightly packed with chips. Vibration was easily dealt with by simple measures such as mounting a plate to hold the Warp cells in place, and mounting the rack holding the Warp machine with shock-absorbing mounts.

As the first full-scale prototypes were being built, we began to look forward to the production Warp machines. The change from wirewrap to printed-circuit boards allowed some redesign; in particular, we simplified the cell-to-cell communication, freeing a lot of board area. This was used to expand the cell data and program memory and add local address generation and local control to block a cell if it tried to write to a full queue or read from an empty one. This created a much more powerful cell, with flexibility comparable to a standard computer (o).

Apply originally was developed by L. Hamey as a C subroutine package for writing image processing functions; the programmer would write a simple subroutine that processed a window of an image and the Apply subroutine would "apply" the subroutine all across an image. This was done to speed up image processing using a subroutine package for accessing images in different formats. Hamey adapted the C subroutine idea to a code generator for the Warp machine (m) that took W2-like Apply programs and generated W2 programs. I-C. Wu developed the first "full" Apply compiler (n) that took Ada-like Apply programs and generated W2 code [4]. This compiler took advantage of the Warp cell's capabilities and generated efficient W2 programs for local image processing functions. Simultaneously, Hudson Ribas developed a library of approximately 100 Apply programs [16].

The new Warp prototype was used extensively with Terregator in stereo vision, obstacle avoidance (using the ERIM scanning laser rangefinder), as well as color-based road following (l).

Navlab work on the Warp machine (l) included development of a geometry module for color road classification, which was tested from the beginning on the Warp machine. One of the wirewrap prototype machines was mounted in Navlab (o), with the overall system structure shown in Fig. 2. We replaced a complex linking procedure that combined the C program calling the W2 Warp program with a runtime code downloading interface (n). The same interface supported a remote procedure call of Warp routines over the Ethernet, which greatly aided development of the Warp code. We soon demonstrated color-based road following and ERIM-based collision avoidance. Both the color and the ERIM code were run on the same Warp machine; we thought we could get better performance with two Warp machines, and tried this idea later in the year (r), when we also seriously addressed the issue of cooling for the Warp machine, particularly its external host. We installed a special air conditioner for Warp, and added thermostatic cutoffs. This allowed us to run the Warp machines continuously on Navlab and allowed us to use them remotely when the Navlab was at home and connected to Ethernet.

A second, smaller (four-cell) production Warp system was mounted on Navlab (r). With two separate Warp arrays we could do the ERIM processing in parallel with the color-based road following. However, there were serious problems with mounting a second Warp machine on Navlab. The Warp cells themselves were not a problem; a ten-cell array could easily be split into two arrays. But the external host boards had to be duplicated, at considerable expense. Moreover, the external host was one of the least reliable components; duplicating it reduced its reliability correspondingly.

The major application of the Warp machine from here on was color-based road following. An adaptive color classification algorithm, using one or two cameras (one with

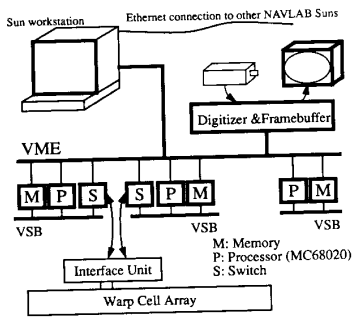


Fig. 2. The Warp architecture on Navlab.

the iris wide open and one with the iris nearly closed, to increase the dynamic range) and a simple geometric model was implemented (s). The SCARF road-following algorithm was re-implemented on the Warp machine with a speed of 4 s (later, 2 s) per image. In the resulting system, Navlab was driven at 1 m/s, with a processing speedup of 10–100 over the same algorithm running on the Sun. SCARF speed was still further improved later.

ALVINN, a neural net-based road-following system, was developed next (u) [13]. A three-layer neural net was trained to recognize driving direction in graphics-generated road images. The training was done off-line, in an 8-h run on the Warp machine in the laboratory. The resulting trained network was then used to drive Navlab. Images could be processed as quickly at 0.75 s/image; a new Navlab record of 1.3 m/s was set. Later it was found that we could train the network "on the fly" by feeding it live road images and driver steering angles while Navlab was under human control. This training was done using the Warp machine on the Navlab. The resulting technique was very powerful; we could, for example, train the network by driving the Navlab halfway along a test course under driver control, and then allow the network to take over vehicle control.

In order to see further speed improvement both in the color-based and the neural net-based road following work, it was thought that the Sun 3 Warp host should be replaced with the newly available Sun 4. The Sun 4 was within a small performance factor of the Warp for ALVINN, in which careful coding could avoid floating-point computation. Moreover, the Sun 4 required less power, space, and cooling than Warp, all of which were critical limitations on Navlab. The Sun 4 was also easier to program and more reliable. Accordingly, the Warp machine was taken off Navlab (v).

III. FIDO

FIDO was a stereo-vision navigation system used for the control of robot vehicles; it included a stereo-vision module, a path planner, and a motion generator. This system descended from work done by Moravec at Stanford [11]. After Moravec came to Carnegie-Mellon in 1980, work was done by C. Thorpe and L. Matthies [10], [15], who gave the system its name. More recently, work was continued by G. Klinker, J. D. Crisman, and E. Clune [2] and others. This vision system was unusual in its longevity and in the range of speed over its span of development: Moravec's original algorithm, which was heavily optimized (though different in many important ways from the FIDO algorithm), took 15 min to make a single step while running on an unloaded DEC KL 10; the Vax 780 implementation rat at 35 s/step; the Sun 3 implementation took 8.5 s/step; and the implementation on the Warp machine took 4.8 s/step.

A. FIDO Algorithm

FIDO was a feature-based algorithm. A *feature* was a point that was detected with FIDO's interest operator and located in three-dimensional space by correlation between the left and right images. An *obstacle* was a feature that the vehicle could not drive over, i.e., a feature sufficiently above ground level. It was assumed that all actual obstacles to the vehicle would have enough image features to be detected by FIDO as obstacles.

FIDO performed the following steps, as shown in Fig. 3. First, it took two 512×512 images of its environment, a left image and a right image. These two input images were reduced by the *Image Pyramid Generator* by successive factors of two, creating images of size 256×256, 128×128, and so on. Then *Image Pyramid Correlation* was used to locate all of the previously known features in the new right image. If the feature could be seen in the new scene, it became a tracked feature. *Image Pyramid Correlation* was then used to identify the tracked feature from the new right pyramid in the new left pyramid. Once the corresponding features were located, the three-dimensional position of the obstacle creating that feature was identified and the vehicle was directed one step toward its goal by the *Path Planner*. The *Interest Operator* picked new features in the right pyramid so that new obstacles moving into the scene could be detected. Again *Image Pyramid Correlation* was used to find the corresponding point features in the left image pyramid. The new features and the tracked features were combined to form the new list of "previously known" features for the next image.

B. Implementation of FIDO on Warp

FIDO was implemented on the demonstration Warp system as well as the prototype Warp machine. In the summer of 1984, P. Dew (on leave from the University of Leeds, England), C. H. Chang, Matthies and Thorpe designed a new version of the FIDO system to run on Warp, which was then in its initial design phase. They identified the three major vision algorithms (correlation, interest operator, and pyramid generation), which were considered to be suitable for implementation on a systolic array such as the Warp machine. Then they redesigned FIDO to run on the Warp machine. Next the three vision modules were implemented using Warp microcode by Klinker on the demonstration Warp system. Later, when the prototype Warp machine was available, the modules were reimplemented by Clune using W2 and the external host and the Warp array ran parts of the algorithm in parallel.

Each of the modules that were implemented on the Warp machine will now be described, and their performance will be given.

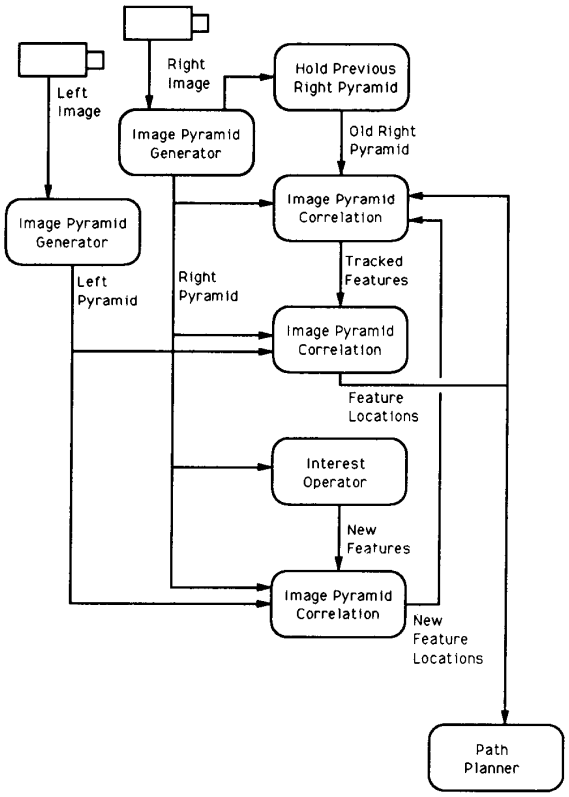


Fig. 3. FIDO block diagram.

1) Image Pyramid Generation: The image pyramid consisted of seven levels, starting with a 512 × 512 image and ending with an 8 × 8 image. Areas of 2 × 2 pixels were replaced by one pixel in the next level of the pyramid. The new pixel value in a lower resolution image was computed by averaging over a window in the higher resolution image. The simplest averaging was to take a 2 × 2 pixel area and average it to one pixel. The first implementation on the Warp array used overlapping 4 × 4 windows, which gave slightly better results than 2 × 2 windows.

The pyramid generation algorithm was implemented in Warp assembler language in a systolic scheme, as suggested by H. T. Kung for convolution-type algorithms [6]. The algorithm accumulated 16 pixels in a 4×4 window and then normalized to produce one reduced pixel value. This was mapped onto the Warp array as nine modules, with the first eight each adding two new pixel values to the accumulated partial sum, and the ninth module normalizing the result. The second, fourth, and sixth modules also stored the partial results until the necessary pixels from the next row underlying the 4 × 4 window had arrived at the module. The new data and the partial results were then sent together to the next module.

A simpler sequential algorithm (with nonoverlapping reduction windows) took about 1 s on a Vax/780. Nine Warp cells provided a speed-up of 14 over the Vax, which was relatively small. The implementation of the pyramid generation algorithm was communication intensive: it used the adder effectively only half of the time (in every other row). It did not use the multipliers at all (except for a normalization). Each Warp cell was used as a 2.5 MFLOPS machine, for 25 MFLOPS from the array. This explains the relatively small speed-up of the pyramid generation algorithm. Adding more cells would not increase the speed since this would not reduce the communication requirement.

This module was later reimplemented as a C program to run on the cluster processors, since very little computation was done here. This made it possible to do the two pyramid generations in parallel using the two cluster processors. In this implementation nonoverlapping 2×2 windows were used instead of the overlapping 4×4 windows in the implementation on the Warp machine, to simplify the computation.

2) Interest Operator: FIDO detected features with an interest operator, which was designed to detect points that could be localized well in different images (for example, corners). Such points had image intensities that changed rapidly in all directions. The interest operator took squared pixel differences in the 3 × 3 neighborhood around the point [15]. The output of the operator was the minimum of the squared differences in the vertical, horizontal, and both diagonal directions. The interest values were locally maximized in 100 subimages that were arranged in a 10×10 grid. The maxima of all subimages were stored in a list ordered by decreasing interest values. This gave a set of point features, distributed across the image, that could be localized in other images.

Only the first part of this algorithm, accumulating squared pixel differences in all four directions for every pixel, was implemented in Warp assembler language. In the demonstration system, the processing stopped here. Later, we implemented the minimization, maximization, and list formation on the cluster output processor.

The interest operator did not offer a good partitioning into modules with similar timings. We thus did not try to implement it in a systolic scheme, as was described for the pyramid generation algorithm in the previous section. Instead, we used the *input partitioning* model [9] where data were divided into equally sized parts. In this scheme, each cell performed the complete algorithm on a portion of the data. An $m \times n$ image was divided into c vertical stripes to be processed on c different cells. For the interest operator, the stripes had to overlap by four pixels, due to the width of the operator window. Thus, every cell ran on $m \cdot \left(\lceil \frac{n}{c} \rceil + 4\right)$ pixels. The systolic communication facilities were then used like a "bus": each cell received data from the previous cell and sent it to the next cell. The host sent the data interleaved such that each cell could use every cth pixel for itself. At the beginning of every new iteration, c new pixels were sent over the "bus." The offset between programs that ran on neighboring cells was two cycles so that each cell started a new iteration exactly when a new pixel arrived.

The sequential algorithm ran in about 2.65 s on a Vax/780. Ten Warp cells provided a speed-up of 26.5. The adder was the most used resource of the interest operator. It was used in 40 out of 65 cycles of the innermost loop. The multiplier was barely used (four multiplications in 65 cycles). The algorithm thus used each cell as a 3.4 MFLOP machine. The addition of

more cells would greatly improve the speed. In the described implementation, each cell needed a new pixel every 65 cycles. Thus, maximally 65 cells could have been used in parallel before the interest operator had become I/O limited.

3) Image Pyramid Correlation: For a given pair of images and a given list of point features in one image, the correlation algorithm found the corresponding point features in the other image. The search for the most likely correspondence was performed on the image pyramids, starting at the lowest resolution (8×8) image. At each level, a 4×4 template around the interesting point was correlated with an 8×8 search area in the other image at the same resolution. The best matching position of the template in the search area determined the position of the search area in the next higher resolution image in the pyramid [11].

A pseudo-normalized correlation was used, as given by this formula:

$$\mathrm{CORR}_{lm} = \frac{S_t - t_{\mathrm{mean}} S_1}{t_{\mathrm{var}} + (S_2 - S_1^2)/16}$$

with

$$S_1 = \sum_{i,j=0}^{3} I_{i+l,\,j+m}$$

$$S_2 = \sum_{i,j=0}^{3} (I_{i+l,\,j+m})^2$$

$$S_3 = \sum_{i,j=0}^{3} t_{ij} I_{i+l,\,j+m}$$

where t_{ij} denotes the template element at position (i, j), and $I_{i+1,\,j+m}$ denotes pixel at position $(i + l, j + m)$ in the image.

In the Warp assembler language version of the algorithm, the Warp machine found the positions of all features for one given pyramid level at a time. First, templates for all pyramid levels were sent. The cells stored the templates and computed their means and variances. Then the search areas of each level were given to the Warp array in the same sequence as the templates. The cells correlated the current template with the current search area and sent the correlation results for every template position to the output cluster. The cluster processor then found the best position of each template within its search window and determined the search areas for the next higher resolution. The process was repeated for all of the images in the pyramid.

The correlation algorithm was implemented in a systolic programming scheme, just as in the pyramid generation algorithm. It was designed as nine modules. Each of the first eight modules covered two template elements. The algorithm was designed so that initially each module received the template elements and stored the respective template elements of each template. The mean and the variance of all templates were computed and stored in the ninth module. Then, in the correlation phase, each module got the pixels of the search areas and the partial sums S_1, S_2, and S_t from its left neighbor and updated the partial sums before it sent them to its right neighbor with the next pair of pixels. As in the case of data pyramid generation, the second, fourth, and sixth modules stored the derived partial results until the pixels of the next row, underlying the current window position, arrived. The ninth module combined the partial sums and the mean and variance of the current template into a correlation value that denoted how well the template fitted the data in the search area at the current position.

The sequential algorithm took about 2.3 s on a Vax 780. Nine cells provided a speed-up factor of 78. This was a much higher speed-up than that achieved by the pyramid generation algorithm and the interest operator because the multiplier was used in every cycle and the adder was used in every other cycle. Each cell thus ran here as a 7.5 MFLOPS machine. The communication facilities were also used in every other cycle. Therefore, the correlation algorithm was a fairly well balanced algorithm. The maximum speed-up would have been reached if 18 cells had been used (due to communication requirements).

This module was originally written as a systolic program, but could not be reimplemented in W2 in this way because the prototype W2 compiler allowed only homogeneous code. Instead, it was implemented using input partitioning, like the interest operator.

C. Performance of the Vision Modules

The reimplementation of FIDO led to a total system time for one step of 4.8 s, which was a large speedup over the original time, but still relatively small compared to the time that had been achieved by that time on a Sun 3 alone (8.5 s). In this section we will analyze the performance of the FIDO system on the Warp machine.

Most interesting was the pyramid generation module on the Warp machine. It actually took longer to run on the Warp machine than on the Sun alone. This was because the data flow between the clusters and the Warp machine was unbalanced. Time-consuming manipulations were required to order the data correctly for the Warp machine in this implementation, but the actual pyramid generation on the Warp array was not computationally intensive. The array was virtually starved for data. This was a case where the ordering of data was too complex for the Warp machine (specifically the clusters). A more efficient implementation would be for the cluster processors to send the pixels in the order that they were stored in memory so that data could flow rapidly into the array, and the Warp array could reorder the data.

The interest operator and correlation functions did not perform at the predicted speeds on the prototype machine, although they were faster than the comparable Sun functions. If the startup times on the Warp machine were subtracted (the startup time was much lower on the production Warp machine), then the actual times were close to the predicted times.

The interest operator required about 0.1 s of Warp array processing time for the ten-cell implementation compared with a 1 s Sun 3 time. Additional time was spent starting the Warp array (about 25 ms). However, most of the time was spent in post-processing. After the interesting operator was run, the cluster processors sorted and selected the resulting data. This

was about 28% slower than the Sun 3 processor, because of a slower clock rate.

The correlation function had less than a factor of three speedup compared to a Sun 3 alone. As with the interest operator, the time required for the correlation function on the Warp array was small. However, the time spent processing data for the Warp array on the cluster processors dominated the total execution time. This time included the following:

- Startup overhead of 25 ms. In one step, correlation was called seven times, for a total overhead of approximately 0.2 s.
- Rearranging data for the Warp machine. Complex addressing was needed to send the image patches from the different pyramid levels to the Warp machine.
- Fixed loop function of the W2 compiler. A fixed number of features must be processed in every correlation, in our case 50, although the average number of features in a correlation was approximately 25.

Work on FIDO stopped in 1987. The move from Terregator to Navlab, with its ERIM laser range scanner, ended it. FIDO's stereo vision was not as reliable and could not be made significantly faster than the ERIM scanner. While FIDO could locate a small number of "feature points" in a few seconds of Warp machine time, the ERIM scanner provides a dense three-dimensional array of points in one-half second scanning time and a few seconds of processing. Moreover, ERIM worked much more reliably than FIDO—it could even be used at night, and FIDO's interest operator, designed to look for object corners in indoor images, never performed very well outdoors; it was confused by image clutter, such as leaves, in outdoor images.

IV. SCARF

Supervised Classification Applied to Road Following (SCARF) combined color and texture information with a geometric road model to recognize the road and drive Navlab. The system labels every pixel in an image as road or off-road depending on how well the color of the pixel matches road and off-road colors from previous images. The road location is determined by matching an ideal road shape model with the labeled image data. This location is then used to update the stored road and off-road colors and to steer the robot vehicle.

SCARF has had several implementations on the Warp machine. The first of these implementations was written in W2 and Apply on the prototype Warp machine. This implementation showed only a factor of two speedup over the Sun 3 version of the code. Later versions of SCARF were implemented on the production machine. We used the Warp machine to process two larger images rather than the one smaller image of the prototype implementation. In this case we saw a speedup of six over the Sun 3 implementation. The fastest SCARF system had a 1 s Warp machine time and a total time of 3 s counting all the overheads, including vehicle control. The speedup was 30 over the Sun 3 implementation for the Warp machine time or ten counting all overheads.

In the next section, we will describe the SCARF algorithm in more detail. We will then discuss how the later SCARF implementations were derived from the first and discuss in general terms the timing of the systems.

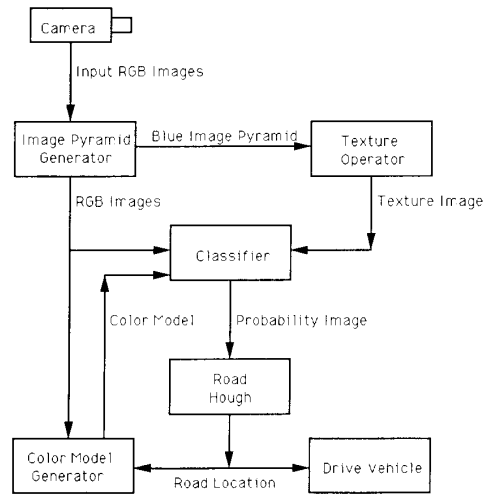


Fig. 4. SCARF block diagram.

A. SCARF Algorithm

The program flow and data transfer between the different SCARF modules are shown in Fig. 4. SCARF starts with (480×512) RGB images from the color camera. The *Image Pyramid Generator* creates an image pyramid for each of the RGB input images. The *Texture Operator* takes the blue image pyramid and creates an image of the texture in the scene. The texture image and the smallest level of the RGB pyramid, the *RGB Images,* are sent to the *Classifier.* The Classifier compares the color of each pixel in the image with stored road and off-road color described by the *Color Model.* Each color pixel is assigned a value in the *Probability Image* representing the likelihood that the pixel is a road pixel. This image is used as voting weights by the *Road Hough* module. Each pixel votes, using its assigned probability, for all of the possible roads that contain that pixel. The *Road Location* with the largest accumulated vote is selected as the best road. The resulting Road Location is used by the *Color Model Generator* to label pixels in the image as road and off-road. The labeled pixels are then used to formulate new road and off-road colors models, which are stored for the next iteration. The Road Location is also used to generate motion commands for the vehicle.

B. Implementation of SCARF on the Warp Machine

In the first implementation of SCARF on Warp, four modules were chosen for implementation on the Warp machine: the Texture Operator, the Classifier, the Road Hough, and the Color Model Generator. The modules were initially implemented by Crisman and Webb for the prototype machine. Later, Chen and Crisman implemented a different version of SCARF on the production machine. This version did more computation than the original system. A final SCARF system was implemented in a single module by Crisman.

The next sections describe in more detail each module that was implemented. We will then discuss how these modules were combined to form the SCARF system and give overviews of their performance.

1) Texture Operator: The Texture Operator measures texture in the scene using the Roberts operator [14], a very simple edge detector. (While the Roberts operator has been superseded for the purpose of detecting the boundaries of smooth regions, its original purpose, it is excellent for detecting the leaf and grass clutter typically found next to roads, its use here.) The operator is applied at two different resolutions on the blue image. First, it is run over the 120×128 image to form a 120×128 *Fine Edge Image.* Then it is run over the 30×32 image to form the 30×32 *Coarse Edge Image.*

The final phase of the Texture Operator is the *Texture Determination* operator. It first creates a *Fine Texture Image* and then counts the fine texture pixels in 4×4 regions to form the smaller *Texture Image.* The Fine Texture Image is computed from the Fine Edge Image, the Coarse Edge Image, and the Average Image. The Average Image is the 60×64 input blue image. The pixel located at row i and column j of Fine Texture Image is calculated by

$$\text{fine\_texture}[i][j] = \text{THRESHOLD}\ (\text{fine\_edge}[i][j]/ (\alpha\ \text{coarse\_edge}[i/4][j/4] + (1-\alpha)\ \text{average}[i/2][j/2])).$$

THRESHOLD is a thresholding function that outputs a 1 if its argument is greater than a particular threshold value and 0 otherwise. The constant $\alpha = 0.2$ is a weighting value; it was set heuristically. The Fine Texture Image detects texture in the input image independently of brightness and scale.

The implementation on the Warp machine used three different modules, the first two of which were Roberts edge operators and the third of which was the Texture Determination operator. Although the algorithm was the same for the edge operators, they were implemented separately since the input images were of different size.

The edge operators were written in Apply, while the last module was implemented in W2. The input images were divided column-wise among the cells. To speed up the processing time, the loops were unwound and each pixel of the output Texture Image was calculated immediately after the calculation of the corresponding 4×4 block of the Fine Texture Image.

2) Classifier: The classification module of SCARF uses a Bayesian classification technique to determine the likelihood that each pixel is a road pixel by matching pixel colors and texture with stored road class colors. A Bayesian classifier takes a d-dimensional measurement vector $\boldsymbol{x}$ and chooses the best class label w_j from a set of K classes, using a previously computed, *class conditional* probability, $P(\boldsymbol{x} \mid w_j)$, for each class [3, Section 2.8]. For our case, $\boldsymbol{x} = [\text{Red Green Blue Texture}]^T$. We assume that the *class conditional* probability can be modeled by a Gaussian distribution and therefore, is totally specified by $\{\boldsymbol{m}_j, C_j, N_j\}$, the mean color and texture, the covariance matrix describing the relationship between the colors and texture, and the number of samples in class w_j. This classifier can be shown to be equivalent to choosing the class that maximizes the following likelihood:

$$\lambda_j = \ln(N_j/N) - d/2 \ln(2\pi) - 1/2 \ln(|C_j|) - 1/2(\boldsymbol{x} - \boldsymbol{m}_j)^T \boldsymbol{C}_j^{-1}(\boldsymbol{x} - \boldsymbol{m}_j)$$

where each pixel provides a four-dimensional measurement vector ($d = 4$). To get the *Probability Image* value, the exponential function is applied to the maximum λ_j.

This module was implemented in W2 by once again dividing the input 30×32 *RGB Images* and Texture Image into column stripes. The input statistical color models were duplicated on each cell. Note that the first three terms of the likelihood calculation can be computed only once for each class rather than once for each pixel. To get the desired probability measure, an approximation to the exponential was used.

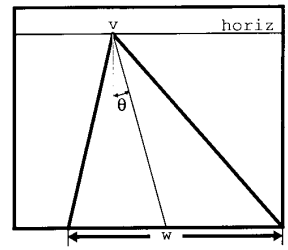


Fig. 5. Road Hough.

Road Hough: This SCARF module searches through all possible road interpretations for the road having the greatest accumulated probability based on the Probability Image from the Classifier. We assume the road is locally nearly straight, and can be parameterized using (v, θ) where v is the column where the center of the road intercepts with the vanishing row in the image and where θ is the angle difference from perpendicular where the center line lies (see Fig. 5). These two parameters are the axes of an accumulator space used for collecting votes. Each pixel in the probability image votes for all the roads that contain that pixel by adding its probability to the proper positions in the accumulator. For each angle θ_i, a given pixel location (r, c) will vote for a set of vanishing points lying between v_s and v_e given from the equations below:

$$v_s = c + (r - \text{horiz}) \tan\theta - (w/l)(r - \text{horiz})$$
$$v_e = c + (r - \text{horiz}) \tan\theta + (w/l)(r - \text{horiz}).$$

where horiz is the horizon row in the image, w is the road width at the bottom of the image, and l is the length from the horizon row to the bottom of the image. The maximum value of the accumulator is chosen to be the road.

This module was implemented by distributing the input Probability Image column-wise among the cells. There was no overlap of the input between cells. However, a complete Hough space was calculated on each cell. To get the output Hough, the individual cell's Hough spaces were added as the output Hough space was passed through the array.

4) Color Model Generator: This function recalculates the road and off-road color models after each image is processed so that the system can adapt to changing illumination conditions. The road and off-road color models are modified in three steps. First a set of pixels in the current image

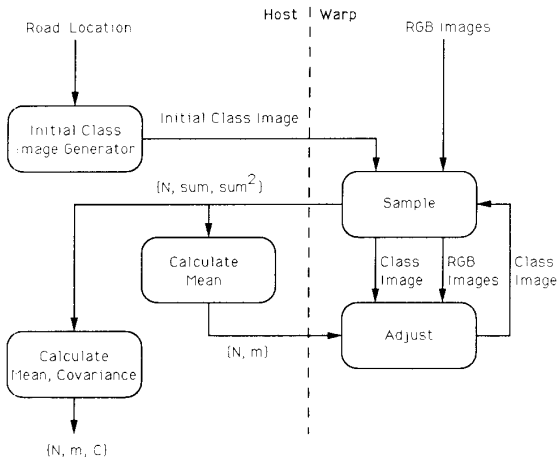


Fig. 6. Implementation of ISODATA clustering on the Warp machine.

is chosen as road and off-road training sets. Next the training sets are subdivided into classes using an ISODATA clustering algorithm. Finally, the new statistical color models are calculated for each class.

The ISODATA clustering algorithm was implemented by a pair of W2 modules; *Sample* and *Adjust*, as shown in Fig. 6. The algorithm used a Class image (a labeled image formed by projecting the predicted road into the image) and the input RGB Images. Sample and Adjust were called repeatedly to converge to the new color models. After the last iteration, the Sample was executed one last time to generate the sums required for the final color models to be calculated.

Sample computed sums of pixel values from a labeled input training set. The labels for each image pixel were stored in the input Class Image which labeled each pixel as one of the road or off-road classes, or as unknown. Sample accumulated the sums $\mathbf{sum}_j$ and the squared sums $\mathbf{sum}_j^2$ of the red, green, and blue pixel value for each class w_j. It also counted the total number of samples per labeled class N_j. Color values were calculated for each class using samples labeled from the Class Image. From this information, the mean color of each road and off-road classes were calculated.

Adjust adjusted the Class Image using the current mean colors road and off-road classes. It read the mean class colors, the old Class Image, and the input RGB Images. Any pixel that was labeled as unknown in the old Class Image remained labeled as unknown. If the pixel was labeled as one of the road classes, then the pixel color (from the input color images) was compared with each of the road mean colors. The pixel was then re-labeled as the class whose mean color most closely matched the pixel color value. Similarly, if the pixel was labeled as off-road in the Class Image, then the new Class Image label was determined by the class whose color mean value closest to the original pixel data. This module wrote a new Class Image, which was input to the next iteration of Sample.

Sample was implemented in W2 by dividing the RGB Images and the Class Image evenly among the cells. Each cell calculated its own partial sum of the color values for each class, partial sum of the color values squared, and number pixels with each label. The resulting sums were accumulated as the values were passed out of the Warp array and to the external host. The host then calculated the statistical color models using these values by the standard statistical equations for mean and covariance. The new mean values were then passed into the Adjust module.

Adjust was implemented in W2 similarly by dividing the input Class Image and RGB Images evenly column-wise among the cells. The mean color values were broadcast to all of the cells. Therefore, each cell produced a column stripe of the output Class Image.

C. Performance of SCARF Implementations

SCARF was chosen for implementation on the prototype Warp machine in January of 1987. At that time, the system was already implemented on a Sun 3 and was processing images in about 30 s/image. During the time that W2 code was implemented, the Sun code was optimized, giving a final Sun 3 time of about 20 s.

The first implementation of SCARF used the prototype Warp machine and was completed in March of 1987. This implementation used only eight of the ten available cells so that the column data could be distributed evenly among the cells to simplify the implementation. Initially, the modules were implemented as described above. However, the time for downloading microcode to the Warp cells and the time required for passing data to and from the cells prevented any speed up of the implementation on the Warp machine over the Sun 3 implementation. This implementation required about fourteen downloads of Warp microcode and image data.

To improve the processing speed, the microcode for individual modules were linked together forming three microcode blocks. As a result, the microcode was downloaded only three times per image. To improve this rate, we implemented the two Roberts operators in W2 to reduce the size of the microcode required for these modules. At this time, the microcode could be linked into two separate sections and then required only two downloads per image.

We also noted that we were passing in input image data repeatedly to the Warp array. By locking the Warp machine so that no other users could access the machine, and declaring the repeated input as global data in the same position in each W2 module, the input could be loaded once, and then used by different modules without being reloaded. With this modification we started to see an improvement over the Sun 3 times.

We implemented a version of Image Pyramid Generator on the cluster processors. This removed some of the load from the Warp array and had the additional advantage that the data from the frame buffer did not need to be copied to the external host before it was transferred to the cluster memory. Instead, the reduction implementation read the first of its inputs directly from the frame buffer and these data were never copied. In order to be as fast as possible, however, this implementation only approximated the averaging that was done in the Sun implementation.

Using microcode linking, common global storage, and optimization of the W2 code, we were able to get the system running in about 10 s/image. This speedup was small, but the results were promising for future implementations on the production Warp machine.

The next implementation of SCARF did more computation than the previous version. It used two color camera inputs rather than the original one camera. It also classified 60×64 images rather than the original 30×32 images. The image pyramid generator now had twice as much data to process, and the classifier and the color model update had four times as much data since the vectors were now six-dimensional rather than four as before. This implementation ran in about 60 s on a Sun 3.

This new SCARF was implemented on the new production machine by Chen and Crisman. On this machine, we had eight times more cell memory for global data and for programs. This machine used DMA for faster I/O from the external host to the Warp cells. The image data was now divided by rows on the cells rather than by columns. This allowed an even division of the 60 rows among ten cells rather than 64 columns divided among eight cells.

The new machine allowed successful use of the reverse data path feature where data could be accumulated on one data path onto the last cell, then the data could be passed back and copied into the other cells. This then allowed us to combine the Sample and Adjust modules from the Color Model Update into one W2 module. To do this the Sample module was modified so that after all of the sums were passed to the last cell, the last cell would calculate the new mean values and pass them back to all of the other cells. Then the Adjust module can be run without the intervention of the external host.

The larger memory of this machine also allowed us to store the RGB Images in a global memory location and then only input this data once per processing step. The increase in program memory allowed all of the modules to be linked into one module, which was downloaded to the Warp machine once. This implementation required about 10 s/image, which was a speedup of six over the Sun 3 implementation.

The last implementation of SCARF on the Warp machine was completed in September of 1989 on the production machine. This version still used 60×64 images; however, it returned to the original one-camera version of the code. The entire SCARF loop was implemented in one W2 function and one function was set up to initialize the whole system. The initialization process read in a 480×512 image and created a 60×64 image in Warp's cell memory. The W2 SCARF loop function started by doing the Color Model Generator on the resident image in memory. Then the new color image was read into the Warp cells and reduced. Next the Classifier and the Road Hough was applied to the input image. Only the resulting road location was passed out of the Warp array. Therefore, the main W2 loop function read only the full size color images, and wrote a couple of floating point numbers representing the road location in the image. The road model colors were stored completely internally in the Warp array.

This version of SCARF ran in 1 s of Warp machine time and a total of 3 s of time, which includes some limited displays and sending motion commands to the robot vehicle. This implementation is challenged only by a similar Sun 4/Androx implementation that processes lower resolution 30×32 images in 3.5 s.

V. ALVINN

Autonomous Land Vehicle in a Neural Network (ALVINN) applied connectionist techniques to the same problem addressed by SCARF, that is, road following using color images [12]. The key difference is that while SCARF was "trained" by hand, adapting standard vision algorithms to the recognition of a road, ALVINN used a neural network learning algorithm to automatically learn what image features were useful to discover the position of the road.

The development of ALVINN on the Warp machine went through two phases. In the first phase, from approximately February through May 1989, a road image generator was implemented and used to generate training images that were fed, together with the correct road position, to a standard back propagation learning algorithm. The back propagation algorithm ran on Warp, off-line; training runs were done overnight in the lab. After training, the learned network was used on Navlab to control the vehicle. (The network was run on a Sun 3, since application of the network, once it was learned, required relatively little computation).

This training technique fully exploited the power of the Warp machine; 8 h runs were used of the Warp machine, with approximately three-quarters of the time during these runs representing actual Warp machine time. Comparable training on a VAX 780 would have taken months.

Training using this method demonstrated the feasibility of using a neural network to control a robot vehicle. But the method suffered from a serious problem. Essentially, the process of adapting computer vision techniques to road recognition was replaced by the process of adapting computer graphics techniques to road image generation. This "forward" generation problem was easier than the "inverse" recognition problem, at least for the simple roads in the park, but it still required human intervention, so that the generated road images accurately represented the range of images that would be presented to Navlab. For successful navigation in more varied environments, the road image generation code would have to become more and more complicated and difficult to program and test.

To overcome this, training "on the fly" was attempted, starting in June 1989. Road images were taken directly from the camera, subsampled, and presented to the neural network together with the current driver's steering angle (with the vehicle under human control). By shifting the road image and steering angle, many example images could be created from one. With the Warp machine on Navlab, back propagation was used to modify the neural network weights as the vehicle was driven up the road.

Remarkably, it was found that with a short sequence of a few tens of images, the network could be trained successfully to follow the road. The 8 h runs on the Warp machine in the lab were replaced by short runs driving Navlab for about

10 min. Apparently, the intense training of the network in the long runs was unnecessary; in fact, the road-following problem was much easier than it had appeared based on the long runs.

VI. Evaluation of the Warp Machine on Navlab

We now critically evaluate the Warp machine in light of the Navlab experience. We will treat hardware and software separately.

A. Warp Hardware

The Warp hardware consists of three components: the Sun host, the external host, and the Warp array itself.

1) The Sun Host: The choice of a Sun as the host of the Warp machine was one of the good early decisions made in the Warp project. At the time, the Sun workstation was one of the most powerful general-purpose workstations available; as it turned out, Sun continued to lead the field both in hardware and in software. The Navlab group decided to use Sun workstations as the basic general-purpose computing element on Navlab. Since the Warp machine had a Sun host, Navlab programs could be run on the Warp host whether or not they used the Warp array. This was important because of the limited space and power on Navlab; having to provide power and space for a workstation that could only be used for controlling Warp programs would have been an extra burden. This is demonstrated, in fact, by the eventual decision to remove the Warp machine from Navlab; this happened largely because the Navlab group moved to Sun 4 workstations. Upgrading the Warp host to a Sun 4 would have required extensive changes to the software. Thus, the Warp host would have been unavailable for running the rest of Navlab software.

2) The External Host: The Warp machine's external host consists of three MC68020 processors in a VME card cage, together with their memories totalling 14 Mbyte. Two of these cluster processors input and output data to the Warp array, through a special board called the switch. The third processor performs auxiliary functions; it is called the "support" processor. The external host communicates with the Sun through a VME bus repeater. The external host card cage also held commercial digitizer boards, which were originally Datacube and later Matrox VIP boards.

A key early decision was to use a commercially available system that was programmable in C. This decision has been validated by the ease of code generation by the W2 compiler for the external host input and output routines (using a commercially supplied C compiler) and by the availability of commercial boards for digitization. We had to do little software development for the basic functionality, and hardware development was limited to the switch board.

Because of the use of industry-standard processors and busses, the external host was the weakest part of the Warp machine. In our early versions of FIDO on the Warp machine, this kept us from realizing full use of the Warp array because of the constraints in rearranging data on the external host.

The decision to include the support processor was questionable. The support processor was intended to be used for auxiliary functions, such as controlling the digitizer board and possibly controlling the driving functions on Navlab; as it turned out, the digitizer boards were controlled by the Sun, and Navlab driving functions were controlled by a separate processor entirely. In fact, the support processor was never used because of the difficulty of programming it and the absence of almost any debugging facility in the external host. System cost could have been reduced by almost a third by eliminating the support processor, with no loss of functionality.

Many of the external host capabilities were completely unused. For example, it was possible to use the external host to drive an RS232 connection; this connection, or another similar standard interface, could have been used to control Navlab. This was never done because the Navlab controlling software and hardware was developed independently and because of the difficulty of adapting such an interface to a new computer like the Warp machine.

Placing the digitizer boards in the external host was also questionable. The intention was to feed data directly from the digitizer boards to the Warp machine under control of the cluster or support processors. In fact, the normal method was to copy data from the digitizer board into the Sun's memory, and then to subsample the data there and pass it to the cluster processor for use in the Warp machine. Only in the most optimized versions of FIDO and SCARF did we actually use the support processor to take data directly from the digitizer board. In all other systems, the data traveled twice over the VME repeater connecting the external host to the Sun.

Surprisingly, most of the reliability problems with Warp came from the commercial external host boards. This happened primarily because the external host boards were more tightly packed with chips than the Warp cells, so they generated more heat and had to be cooled more than the Warp cells.

The large memories in the external host were rarely used in Navlab. Navlab datasets were generally quite small, and there was no need for more than a few megabytes of memory. However, the large memories were useful in order to maintain compatibility with the Warp machines in the lab, where the large memories were used by other programs. When Navlab was docked and connected to the Ethernet, programs could be run interchangeably on the Navlab or the lab Warp machine.

3) The Warp Cell Array: The overall structure of the Warp cell array, a short linear array, has been validated by our experience on Navlab. The linear array was quite capable for the low- to mid-level vision algorithms we intended to implement on it at the start of the project; as the range of applications increased to include mid-level processing in SCARF and the neural network back propagation algorithm, the same linear array was usable. The key reason for this was the very high I/O rate within the array; this allowed us to overcome its limited connectivity.

The short linear array also lent itself to dealing with the relatively small datasets (32×32 or 64×64 images) in SCARF. Our early applications studies of the Warp machine were oriented toward dealing with standard 256×256 or 512×512 images. We thought that the increased power of the Warp machine would make it possible for the same processing then being done on small images to be done on large images, which would improve the accuracy and utility of color vision. As it

turned out, this was not true. No increased vision performance could be obtained by using more spatially dense images; the road was a fairly large object in most of the scene, and where it was small (near the horizon), recognizing it accurately was useless because of other uncertainties in the system. So we turned instead to processing small images with the Warp machine. The short linear array was just as suitable for this as it was for processing large images; in fact, with small images the relatively small Warp cell memory could be used to store previous images and other datasets, as in SCARF.

The two-way I/O pathway within the array was used to allow the computation of a result known to all cells entirely within the array; this was used in SCARF and in some implementations of the back propagation algorithm. In fact, for many purposes a circular connection (allowing the last cell to communicate directly with the first) would have been preferable.

The Warp cell included a hardware floating point; this facility was one of the main reasons for building the Warp machine in the first place, and it was one of the most expensive features in the Warp machine. The Navlab application made good use of Warp's floating-point hardware. In SCARF, the floating point was used extensively in the calculation of road statistics and their application to color pixel classification; in ALVINN, the floating point was necessary for good implementation of the back propagation algorithm. These applications of the floating point came from outside the Warp project; independently, the Navlab group began using statistical methods for color classification, and the neural network group required the floating point for their work. Without the floating point, the Warp machine would have been far less effective as a tool on Navlab.

B. Warp Software

As with hardware, we divide the software discussion into three parts: Warp host (Sun) software, external host software, and Warp cell software (the W2 and Apply compilers).

1) Warp Host: The Warp array was used as an "attached processor" to the Sun. Datasets were downloaded into the external host, and then the Warp array was called to process them, usually while the Sun waited. (In fact, Sun processing could go on in parallel, and this was done in some of the SCARF systems. But generally this feature was not exploited because there was little for the Sun to do from the time the image was captured to the time the road was recognized.)

This model was extremely useful in the development of software for the Warp machine. It was implemented using a mechanism that allowed replacement of a subroutine call in C by a single subroutine call in the Warp software package; the subroutine handled all transfer of data to and from the Warp external host, locking the Warp machine for exclusive use, and downloading and call of the Warp program. This could happen even if the Sun executing the call to the Warp machine was not a Warp host; data would be transferred over the Ethernet to a selected Warp host. It is quite likely that the Warp machine would not have been used much at all in real applications without such a simple method for accessing the machine.

However, the attached processor model implies many overheads. The Sun can become a bottleneck for processing. The startup time for the Warp machine can be quite significant. A serial processor, the Sun, must prepare datasets for a much more powerful parallel processor, the Warp array. Data structures must be moved from the Sun into the external host for processing. All of these overheads seriously affected the performance of the Warp machine on Navlab.

For example, images as captured by the Datacube boards were 480×512 in size. They had to be reduced in size for processing, since spatial resolution was not an important factor in road recognition. This could be done on the Sun, in the external host, or on the Warp machine. Existing libraries of software made image reduction on the Sun trivial—in fact, transparent to the programmer. Programming the external host (the logical place) to do the reduction was difficult, and the Warp cell array could do the reduction only if the images were first transferred from the Datacube boards to the external host memory either by the Sun or the external host. These tradeoffs made image reduction usually happen on the Sun, although in some SCARF systems it was implemented on the external host.

The Warp machine's startup time (time to start up a Warp program with the code already downloaded) was about 25 ms. This time was not a significant fraction of processing time for 256×256 or 512×512 images; in fact, the minimum processing time for 512×512 images was about 60 ms, and usually several times longer. But for small 32×32 or 64×64 images, this time could be a significant fraction of the total time, particularly if several Warp functions were applied to process the image and recognize the road, as in SCARF. As a result, the programmer had to spend a lot of time organizing the Warp functions so they could be executed as the result of a single call, to reduce the overhead. We would have been better served had the startup time been significantly reduced; this could have been done by providing special hardware in the Warp machine's interface unit to allow the Warp machine to initialize itself. As it was, the Sun had to issue special commands to do the various stages of initialization.

Moving data structures back and forth from the external host implied considerable programming difficulty, since many of these structures were embedded in various ways in C programs. This was especially true for FIDO, an old program that had been worked on by a number of programmers. All of the data structures had to be "cleaned up" before the Warp programming could begin; and the process of cleaning up the data structures introduced new overheads.

If we had not used the "attached processor" model, we might have taken an "array-centered" view of the Warp machine. In this model, the Warp array would have been viewed as the central processing resource, and other devices, such as the external host, would have been viewed as supplying data for the Warp machine. We could have attached multiple I/O devices, supplying data from different cameras and perhaps the ERIM laser scanner, and coordinated the processing through the Warp machine. This model would have required much more sophisticated Warp software; we would have needed an operating system on the Warp array to manage all these resources. But such a model would overcome the problems discussed in this section.

2) External Host: The external host software was designed on the assumption that in Navlab speed was of overriding importance, code would be linked together before runtime (i.e., runtime downloading of code was not necessary), and a library of compiled code could be built up that was not changed frequently.

In fact, these assumptions were largely untrue. The W2 compiler made it possible to write new routines and test research ideas using the Warp machine, which made it much more important to allow rapid testing of new code. (The early development of FIDO, with its programming of a few routines in microcode by several programmers over a period of months, much more closely matches the model we had in mind when the external host software was designed.) The difficult programming environment of the external host, which might have been acceptable if the code was not modified much, instead meant that it was reduced to performing I/O to and from the Warp array, using programs generated by the W2 compiler. And testing of new Warp routines could be best done using an interactive system, which meant that the external host software had to be adapted to allow runtime downloading of code.

One of the reasons for the difficult programming environment on the external host was that control of the development of external host software was transferred to Carnegie-Mellon's industrial partners at an early stage of the project, well before it was used. This made it difficult to change the software as our applications experience grew. The industrial partners did a competent job of maintaining and extending the external host software as it was originally designed, but the software would have had to be redesigned extensively to be widely used in Navlab.

The use of the external host in FIDO shows its capabilities when the programming difficulties are overcome. Irregular operations were mapped onto it as part of pre- and post-processing of data from the Warp machine. Also, it performed memory access-intensive but not compute-intensive computations as well as or better than the Warp array, which could also allow the Warp array to be used for something else in the meantime.

3) Warp Array: In this section we discuss the Warp array software (primarily the W2 compiler) as seen by the user. This includes many design decisions that were essentially forced by the Warp cell hardware, and thus are really hardware issues. Distinguishing between W2 issues forced by the hardware and forced by other concerns is appropriate for a paper on the W2 compiler, not an applications paper.

W2 made the Warp machine much more programmable than we expected. This led to major changes in the importance of some parts of the system and made it possible to overcome deficiencies in one area by using another instead. For example, we could modify our programs to accommodate a regular data pattern from the host, which led to higher I/O rates. This was important even in the later versions of the host, which had faster processors and higher data rates, but which could use DMA, which required a regular address pattern. This flexibility was the main reason we were able to observe the predicted performance of FIDO in actual Warp runs.

W2 is a simple "Pascal-like" language for programmers to implement. All that is required is that the programmer understand a very simple model of the machine, i.e., that there are 10 cells in parallel connected by a data path. The programmer, however, must decide how to parallelize his programs. A W2 function to average a 4×4 window of pixels is as follows:

```
procedure reduce ();
begin
    int r, c, row, pos;
    float acc;

    for r := 0 to eval(SWATHROWS-1) do begin
        for c := 0 to eval (NCOLS-1) do begin
            pos := 4 * r*IMGCOLS+c * 4;
            acc := imgbuf[pos]+imgbuf[pos+1]+
                   imgbuf[pos+2]+imgbuf[pos+3];
            pos := pos + IMGCOLS;
            acc := acc+imgbuf[pos]+imgbuf[pos+1]+
                   imgbuf[pos+2]+imgbuf[pos+3];
            pos := pos + IMGCOLS;
            acc := acc+imgbuf[pos]+imgbuf[pos+1]+
                   imgbuf[pos+2]+imgbuf[pos+3];
            pos := pos + IMGCOLS;
            out[r*NCOLS+c] = pos * 0.0625;
        end;
    end;
end;
```

Before this function is called, the input image to be reduced is divided into row swaths across the cells. "Pos" marks the position in the input image and "acc" accumulates the sum of the pixel values.

W2 made it possible to experiment with different algorithms, in the context of a research system such as FIDO, while getting good use of the powerful Warp array. As we programmed more and more of FIDO on the production Warp machine, programmability was essential, especially as it allowed us to make use of more complex programming models that used the powerful Warp array more and required less intervention by the relatively weak host.

Apply was used far less in the Navlab work than we had hoped. This was partly because of the relatively late introduction of Apply into the project; the first true Apply compiler for the Warp machine was not running until the fall of 1987, one year after the Navlab group began working with the Warp machine. By this time much of the programming difficulties Apply addressed had been overcome by learning on the part of Navlab programmers. Just as important, Apply code tended to be larger than W2 code for the same problem. In order to process the borders of the image properly, Apply duplicated the inner loop of the image processing function once, leading to a doubling of the code size. This was a serious problem when the user was attempting to keep all code for the entire SCARF application, for example, on the machine at the same time. W2 programs were smaller, though no faster than the Apply programs.

Border processing was a problem for the W2 programs, too,

however. The C functions processed borders by duplicating rows and columns near the edges of the image. This was hard for the W2 programs to do. It was simpler just to use a constant value (0) for the border of the image. However, this led to spuriously high values of edge detectors like Roberts near the image border. This sometimes affected the accuracy of the road image processing based on the texture image.

W2 did not support function calls until quite late in the project. Macro calls were used instead. This lead to code size problems for transcendental functions, so that approximations were used instead. This often lead to less accurate results than those used on the Suns. In particular, the classification in the SCARF system was often noisier for the W2 implementation than for the Sun implementations.

There were some peculiarities in converting data between the external host and the Warp array. Because the Warp machine's primary processing power was in the floating point, all images were best processed in this way. The interface unit had hardware conversion of 8- and 16-bit integers to and from the floating point, but could not convert 32-bit integers. As a result, 32-bit integer images in the external host had to be treated differently from 8- or 16-bit integer images.

Primarily as a result of the fixed-size queues, it was impossible to use a variable length for loops in W2 programs until the design change that allowed blocking on writing to a full queue, or reading an empty one, was fully integrated into W2. This meant that image sizes were fixed at compile time. This increased the code size (for example, in SCARF there were three different versions of the Roberts operator).

It was impossible within W2 to execute a W2 function repeatedly until some condition was met, for example, repeatedly classifying image regions until convergence was achieved. This was a consequence of the distributed nature of control in the Warp machine. The result of this was that the Warp host was involved in repeatedly calling W2 programs and testing for convergence, which significantly increased overheads.

Partly as a result of the small datasets used on Navlab, and partly as a result of limited hardware support, we still needed to use speed tricks to generate Warp code that was significantly faster than the Sun code. For example, we had to avoid using division on the Warp machine, especially when doing integer index calculation for arrays.

Given the research that was going on in the Warp project while the Warp machine was being used in the Navlab project, it is remarkable that things worked as well as they did. Navlab programmers commonly had to deal with new features in the W2 compiler, for example, and it is only due to the good support from the compiler group that they were able to overcome bugs that were due to idiosyncrasies of the machine and were extremely difficult to identify without experience.

VII. Conclusions and the Future

This paper tells the story of a unique experiment: the installation and use of a parallel supercomputer on a robot vehicle. Let us try to summarize and draw some conclusions from this experiment.

- The Warp machine was useful in the Navlab project. The programmable floating point capability it brought to Navlab was unavailable by other means. Key elements of the architecture, such as its high I/O rate and the short linear array, have been validated by the Navlab experience. The high processing rate of SCARF could not have been obtained without the Warp machine, and it was the presence of the Warp machine on Navlab that led to ALVINN.
- Early applications support is essential. The development of FIDO and other vision applications guided the early design of the Warp machine and provided test programs and early demonstrations. Without this early work, the Warp machine may never have been used on Navlab.
- Continuing software support is essential in a project such as this. It is impossible for hardware and software designers to anticipate all of the issues that will turn up in use of the machine, even if applications designers participate early in the project. This is partly because applications can change, and partly because success in one area can affect others. For example, the Warp machine became much more programmable than we anticipated because of the W2 compiler, which made the design of the external host partly obsolete.
- The "attached processor" model used in the Warp machine is natural and easy to use by the programmer, but it leads only with great difficulty to large speedups in programs. Data structures must be redesigned, careful attention has to be paid to small details of implementation, and so on. If we want to see speedups more than a factor of about ten, we must abandon this model.

Based on our experience, we can make these recommendations and predictions about computer systems and their software in mobile environments like Navlab:

- For tasks like road following, computing power on the order of the Warp machine (100 MFLOPS) is sufficient. A simple calculation based on the Navlab experience shows this: in a road-following task, we can assume that less than 1000 operations/pixel are needed, and on image resolution of about 64×64 or 128×128 should suffice. Processing more than a few frames per second is unnecessary since vehicle dynamics are slow. Multiplying these requirements, we get a total computing power needed of 40–100 MFLOPS.
- Floating point is a requirement in such a processor. A single precision floating point makes programs far easier to write, eliminates sources of bugs, and allows easy implementation of mathematical techniques like the SCARF classification algorithm, which used transcendental functions.
- The biggest limitation in Warp performance was the interface between the digitizer boards and the Warp array. The image had to traverse the VME bus (and possibly the VME bus repeater and Sun internal bus, twice, if it was being transferred under control of the Sun instead of the external host) at full resolution and under program control. This severely limited Warp performance.

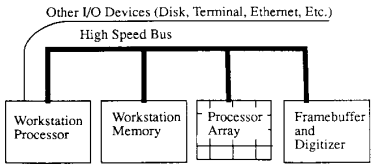


Fig. 7. Possible future robot vehicle system.

In future systems, the frame buffer must be tightly integrated, and provision should be made for subsampling (preferably using averaging with overlapping windows) at the framebuffer. A fast interface to a display would help in debugging and understanding system behavior.

- Mobile vehicles have limited power, space, and cooling; this was the issue that directly led to Warp's being removed eventually from Navlab. Higher levels of integration must be used in future systems to reduce these.
- Another important issue is the quality of the programming environment. Debugging facilities for Warp were almost nonexistent, and programming was difficult. Future systems should include debugging facilities, at least at the level of breakpoints at the source level, and the inspection of program variables.
- Program generators like Apply can make a significant contribution to making such a system usable, but they must be introduced early in the project so that the advantages they offer, mainly easier programming and learning, can be used.
- Any such computer system must be fairly general-purpose to be useful. In a research environment, it is difficult to predict which applications will be most important in a year or two—developments outside the project can overtake certain approaches (as the development of the ERIM scanner made FIDO obsolete). The computer system must be flexible enough to be applied to different tasks as requirements change.
- No experimental computer system can remain at the leading edge of the field for long. When we started the project, Warp's programmable 100 MFLOPS were well ahead of what was available commercially; eventually, serial processors (nearly) caught up.

There are two possible system configurations that follow from the observations above. One, shown in Fig. 7, is a modified version of Fig. 3, the system configuration used on Navlab. Essentially, we couple an array of Warp-like cells (called a processor array) more tightly into a workstation, using a bus with a usable bandwidth of 10–20 mbyte/s (instead of the VME bus with its effective limit of 1–4 Mbyte/s) under program control by the workstation. We could also place a framebuffer with display and digitizer on the same bus in such a system. The high bandwidth on the bus would allow us to overcome the bottlenecks in image I/O in the Navlab system.

Such a system would maintain the attached processor model, but would attempt to overcome its problems by using a high-bandwidth bus and coupling the processor array more tightly with the workstation. In such a system, a tightly integrated programming environment between the workstation and the processor array would be essential to overcome the programming bottlenecks in moving processing from the workstation to the processor array. For example, there should be a common name space of variables between these two components.

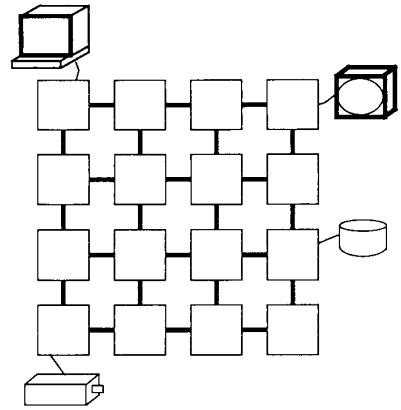


Fig. 8. Another possible future robot vehicle system.

An alternative system would eliminate the workstation entirely, as shown in Fig. 8. We could build a processor array and attach frame buffers and disks as well as terminals for user interaction. In such a system, we overcome the limitations of the attached processor model directly by changing to an array-centered view. But we would have to overcome significant problems in operating system design, resource allocation and management, and so on to use such a system. However, the potential performance of such a system would be much greater, since there is no serial bottleneck whatsoever.

Acknowledgment

"We" in this paper refers to a very large group of people, indeed—too many to list here. The Warp project at Carnegie-Mellon and General Electric and Warp applications development (including the Parallel Vision and Navlab projects) included over 70 people. We also benefited from the support of the Field Robotics Center and from many discussions with the vision group at Carnegie-Mellon. In addition, we would like to thank Kluwer Academic Publishers for permission to publish this paper.

References

[1] M. Annaratone *et al.*, "The Warp computer: Architecture, implementation and performance," *IEEE Trans. Computers*, vol. C-36, no. 12, pp. 1523–1538, Dec. 1987.

[2] E. Clune, J. D. Crisman, G. J. Klinker, and J. A. Webb, "Implementation and performance of a complex vision system on a systolic array machine," Robotics Institute, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMI-RI-TR-87-16, 1987.

[3] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.

[4] L. G. C. Hamey, J. A. Webb, and I-C. Wu, "An architecture independent programming language for low-level vision," *Comput. Vision, Graphics, Image Processing*, vol. 48, pp. 246–264, 1989.

[5] H. T. Kung, L. M. Ruange, and D. W. L. Yen, "Two-level pipelined systolic array for multidimensional convolution," *Image and Vision Comput.*, vol. 1, no. 1, pp. 30–36, Feb. 1983.

[6] H. T. Kung, "Systolic algorithms for the CMU Warp processor," in *Proc. 7th Int. Conf. Pattern Recognition*, 1984, pp. 570–577; also in *Systolic Signal Processing Systems*, E. E. Swartzlander, Ed. New York: Marcel Dekker, 1987, ch. 3, pp. 73–95.

[7] H. T. Kung and O. Menzilcioglu, "Warp: A programmable systolic array processor," in *Proc. SPIE Symp., Real-Time Signal Processing*, vol. 495, Aug. 1984, pp. 130–136.

[8] H. T. Kung and R. L. Picard, "One-dimensional systolic arrays for multidimensional convolution and resampling," in *VLSI for Pattern Recognition and Image Processing*, K. Fu, Ed. New York: Springer-Verlag, 1984, pp. 9–24.

[9] H. T. Kung and J. A. Webb, "Mapping image processing operations onto a linear systolic machine," *Distributed Comput.*, vol. 1, no. 4, pp. 246–257, 1986.

[10] L. H. Matthies and C. E. Thorpe, "Experience with visual robot navigation," in *Proc. IEEE Oceans '84 Conf.*, Sept. 1984, pp. 594–597.

[11] H. Moravec, "Obstacle avoidance and navigation in the real world by a seeing robot rover," Carnegie-Mellon Univ. Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-3, Sept. 1980.

[12] D. A. Pomerleau, "ALVINN: An autonomous land vehicle in a neural network," in *Advances in Neural Information Processing Systems*, D. S. Touretzky, Ed. Los Altos, CA: Kaufmann, 1989.

[13] ——, "Vision and navigation: The CMU Navlab," in *Vision and Navigation: The Carnegie-Mellon Navlab*, C. Thorpe, Ed. New York: Kluwer Academic, 1990.

[14] L. G. Roberts, "Machine perception of three-dimensional solids," in *Proc. Symp. Topical and Electro-Optical Info. Processing*, J. T. Tippett *et al.*, Eds. Boston: MIT Press, 1964, pp. 159–197.

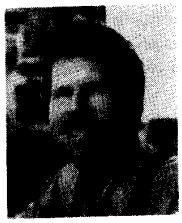
[15] C. E. Thorpe, "FIDO: Vision and navigation for a robot rover," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Dec. 1984.

[16] R. S. Wallace, J. A. Webb, and I-C. Wu, "Architecture independent image processing: Performance of Apply on diverse architectures," *Comput. Vision, Graphics, Image Processing*, vol. 48, pp. 265–276, 1989.

Jill D. Crisman (M'91) received the B.S. and M.S. degrees in electrical engineering from the University of Pittsburgh, Pittsburgh, PA, in 1984 and 1985, respectively, and the Ph.D. degree in electrical and computer engineering from Carnegie-Mellon University, Pittsburgh, in 1990.

She was an active researcher in computer vision on the Navlab project at Carnegie-Mellon University from 1985 to 1990. She is currently an Assistant Professor in the Electrical and Computer Engineering Department at Northeastern University, Boston, MA. Her research interests include computer vision, robotics, pattern recognition, and artificial intelligence.

Dr. Crisman is a member of Eta Kappa Nu, Tau Beta Pi, and the Classification Society of America.



Jon A. Webb received the B.A. degree in mathematics from the University of South Florida, Tampa, in 1975, the M.S. degree in computer science from The Ohio State University, Columbus, in 1976, and the Ph.D. in computer science from The University of Texas at Austin in 1981.

He is currently a Research Computer Scientist in the School of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. His research interests lie in the area of parallel computer vision. He has been responsible for the development of the Apply language and has contributed to the development of the Warp and iWarp computers.