

# An Integrated Environment for Development and Execution of Real-Time Programs

Bernd Bruegge and Thomas Gross

Department of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213

## Abstract

The goal of the Warp Programming Environment (WPE) is to provide easy access to the Warp machine, a parallel supercomputer with a peak performance of 100 MFLOPS that is based on the systolic array architecture. The Warp Programming Environment offers a uniform environment for editing, compiling, debugging and executing Warp programs. It is based on an extensible shell written in Common Lisp and a runtime system written in C. It runs on a network of SUN-3 workstations under UNIX 4.2. This paper describes how the program development environment interacts with the Warp machine to support the development and execution of real-time programs on Warp.

## 1. Introduction

All users of supercomputers are concerned about performance, and in the past, this concern meant that programmers had to be willing to learn about specific implementation details and idiosyncrasies of the target computer. More recently, ease of programming has become an important issue for supercomputer programmers as well. High-level language compilers are probably the most important tool, and – starting with compilers for Illiac-IV [4] – have included more and

---

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

Warp is a service mark of Carnegie Mellon University. UNIX is a trademark of AT&T Bell Laboratories. Sun-3 is a trademark of Sun Microsystems.

more optimizations to remove the need for low-level programming. The execution environment has seen a similar advance to the point that now a multi-user operating system (most often UNIX) is a reasonable choice for a supercomputer [19, 7]. For the Warp machine, a systolic array machine in use at Carnegie Mellon that is capable of 100 million floating point operations per second (MFLOPS), we implemented a programming environment system that combines the runtime environment, user interface (shell), compiler, and debugger to provide the programmer with a uniform mechanism to access a supercomputer.

The recent development of compact and inexpensive mini-supercomputers has led to the use of supercomputers in real-time applications where little or no operating system overhead can be tolerated. (Our definition of real-time application includes all instances where a computer is interfaced to some real-world machinery or sensor. Examples are the use of a supercomputer to control a robot vehicle [18] or real-time image processing. Often the supercomputer is dedicated to a single application at a time; therefore scheduling the different jobs to meet a deadline poses no problem, and a static schedule suffices.) For example, one Warp machine is mounted inside a robot vehicle used at Carnegie Mellon, and another Warp machine is used by the Autonomous Land Vehicle project at Martin Marietta [8]. The need for low-overhead execution often implies a “bare-bones” environment that is not conducive to program development. Therefore an integrated approach is needed that reconciles the different demands of program development and real-time execution.

We report in this paper on the Warp software system and how it supports program development for Warp. Specifically, we discuss how real-time programs are developed using the interactive features of the environment, how they are executed efficiently, and how the environment supports migration between development and execution.

## 1.1. Warp overview

The Warp machine has been described in previous papers [3, 1], and at this time, three Warp machines are in use at Carnegie Mellon. The Warp machine consists of an linear array of programmable processors (called "cells") that is connected to a host system; each cell delivers up to 10 MFLOPS and has local memories for instructions and data. A large number of application programs in diverse domains have been developed for Warp [2, 17, 14]. Here we sketch only those aspects that are necessary to understand the programming issues and refer the interested reader to the other papers. Figure 1-1 shows the configuration of the Warp environment. Each workstation, a SUN-3, runs UNIX and therefore forms a good platform for a process oriented programming environment. Among the processes executing local to each workstation are one or more *Warp shells*. The workstations communicate with a machine called the *Warp host*. This is another SUN-3 which is physically connected via a bus repeater to the Warp array [3]. The Warp host includes two programmable processors called *cluster processors* that act as I/O drivers for the Warp array; these 68020-based processors include a sizable local staging memory to buffer input and output with the Warp array. That is, the Warp array is physically connected as an attached processor to the Warp host system; we will illustrate in this paper that a careful

design of the programming environment can provide the user with an integrated view and a unified model of the system. For this system, the intricate details of the connection between the host and the array are hidden from the user's view.

## 1.2. WPE overview

WPE provides three important features: shared access to the Warp machine, network transparency and efficient management of data and control [5]. Server processes play an important role in their provision. They execute on the Warp host and can be seen as the intermediators between users and the Warp array and external host. One server provides multiple user access to the Warp machine. A second server accepts remote requests to a specific Warp machine. A third server is forked off on the Warp host for every user process accessing a Warp machine remotely.

The use of these servers is independent from each other, providing the user with a wide spectrum of options in accessing Warp. Warp can be accessed remotely from a workstation, or locally from the Warp host. For example, one possibility is shared use of multiple Warp machines; User 1, User 2,..., User m-1 in Figure 1-1 run a Warp shell (Wshell) on a workstation and access Warp interactively from the shell.

**Figure 1-1:** Warp system environment at Carnegie Mellon

For each Wshell, there runs a *user server* on the corresponding Warp host. (Our current implementation allows only access to a single Warp machine at any point in time, but the user can easily switch from one machine to another during the course of a session. The rationale for this design decision is to prevent a single user from tying up all of our Warp machines.) This mode of operation is typical for program development where users access Warp remotely over the network.

Another possibility is accessing the Warp machine for a real-time application, from a program on the Warp host (Direct 2 in Figure 1-1). In this case there is no shell or command interpreter overhead, the user program executes like a batch job without further intervention. Since a program running on the Warp host consumes resources also needed by the user servers, the environment offers also another mode, accessing the Warp remotely with a non-interactive program. Direct 1 of Figure 1-1 provides an example. This mode is used primarily during debugging of the part of the application that does not run on Warp. In this case, the delay introduced by the network can be tolerated as long as the transition to local mode is automatic. All access modes can coexist concurrently in the same environment.

### 1.3. User classes

We distinguish between two target groups of programmers; these groups differ in the kind of support they require from a programming environment. The first group of users accesses the machine interactively. This is the mode of operation that is typically employed during program development. For this group, fast turn-around time and a comfortable user environment are most important. All the tools commonly found on a uni-processor are required to support the activities of this group. As long as supercomputers are not placed in the offices of their respective users, interactive users want at least the illusion of a local machine, instead of having to go to a special place (laboratory, machine room, etc.) to access the system. Users in this group do run different programs during the course of a day and are therefore concerned about program and data downloading time.

The second group of users wants to use the supercomputer in real-time applications. Real-time applications are those uses of the Warp machine that interact with some outside machinery or agent. For example, Warp processes sensor data, controls equipment like a robot vehicle, or evaluates the result data collected from an experiment in real-time. This group of users is willing to tolerate constraints that are unacceptable to the first group, like the need to access the computer from a specific place. On the other hand, these users have more demanding performance requirements and usually cannot accept any overhead in accessing or starting the machine.

Obviously, a programmer will belong to both groups of users in the course of his interaction with a computer, and

might even switch roles a couple times during the course of a day. While a program to control a robot is developed, the usage pattern exhibits the marks of the first user group. When the program is debugged, the constraints imposed by the real-time environment dominate the programmer's concern. The program environment recognizes this development process (and pays special attention to the situation that a problem uncovered in a real-time setting demands that the user returns to the development stage).

## 2. Tools

There are nine major software components in the Warp Programming Environment: An optimizing *compiler*, a symbolic *debugger*, and a *simulator* are tools for the development of Warp programs. The compiler has been described earlier and has been in use for three years [9, 13]. An *editor* (GNU Emacs [16]), a *window manager* (the X window manager [15]), and the *Generalized Image Library* [11] assist in the preparation of programs and support the display of data structures and images. The *Warp monitor* constitutes the runtime system to access the Warp machine. The *Warp shell* is the user interface to the runtime system, compiler, and debugger. It manages the state of the user session and is crucial for the execution of Warp programs. The *servers* provide network transparency for remote applications as well as access to multiple Warp machines in environments where there is more than one machine.

Communication between the components is via the WPE database, which is managed by the Warp shell. Programs for the Warp array are written in W2, a high level language for Warp [3, 9]. A by-product of the compilation is an abstract syntax tree which is accessible by the other components of the Warp Programming Environment. For example, the W2 debugger inspects the syntax tree when it searches for the value of a variable; the Warp shell inspects the syntax tree when it matches the actual parameters of a Warp program call with the formal parameters of the program.

### 2.1. Monitor

Each server communicates with the *Warp monitor*, a software package designed to provide a "virtual Warp machine". The main goal of the Warp monitor is to shield the programmer from the complexity of the Warp array and host system and yet make the hardware accessible. The Warp monitor contains functions for locking the machine, allocation of memory, for the execution and for the inspection of data and code.

In addition to providing a server for the Warp shell, the Warp monitor can also be used by programmers who want to call W2 programs from their own application programs. This includes applications running inside the Warp shell, for example debugging tools for the Warp machine, as well as standalone applications. A package called the *Warp User Package* has been implemented on top of the Warp monitor that enables the application programmer to write standalone

applications using Warp programs from a library without having to know any details about the Warp machine at all.

The Warp monitor allows the user to interrupt or abort the execution of a program on the Warp array, but it does not multiplex different user programs. That is, the monitor supports time sharing but not time slicing. Since numerous programs run only for a fraction of a second or a few seconds, the occasional wait for a longer running program can be tolerated. (This usage pattern is clearly different from the patterns observed at supercomputers placed in central computing facilities, which tend to be used for long running applications. One reason for this difference is the part of our user community that concentrates on computer vision research. Before Warp was available, minicomputers and super-minicomputers were the primary computation engines; programs that run in a couple of minutes on a VAX 11/780 complete on Warp in seconds or less.)

## 2.2. Shell

The Warp shell provides the basic functionality of UNIX shells, such as the C-shell, as well as commands to compile and execute programs on the Warp array. It maintains a set of environment variables such as SOURCEFILE (the name of the current W2 program), WARP (the name of the current Warp machine) and BREAKPOINTS (the set of currently defined breakpoints). These environment variables can be inspected and assigned new values with Warp shell commands.

The Warp shell has several attractive features. First of all, it provides a uniform help mechanism. An example from the help description of a command can be fed to the command interpreter, providing an easy exploration of the command language. Second, it takes care of different levels of sophistication, in particular of the novice and the experienced user; this can often be the same person at different times during the program development. The underlying Common Lisp implementation and the components of the environment are completely hidden from the application programmer; this is useful for somebody who is just interested in using the Warp shell. However, the Lisp implementation and all the software components comprising the Warp environment are easily available if so desired. This makes the Warp shell extensible. In particular, an interested programmer can make use of Common Lisp's powerful control structures to implement new commands.

The Warp shell is designed to run inside a text editor. The advantage of using an editor is that features such as intra-line editing, history buffers, re-execution of commands of previous commands and creation of script files are automatically available without any additional cost.

## 2.3. Debugger

The debugger presents a conventional user model: the user can set breakpoints and inspect variables on individual cells. This model is consistent with the programming model that

demands that the user takes care of computation partitioning onto the array. We found that this model – although simple – is extremely powerful and significantly eases program development for Warp.

The debugger is accessed from the shell level. That is, the commands to set breakpoints, inspect variables, and resume operation are issued like other shell commands. The debugger is line-number oriented. As users explore the debugger, we expect new ways of usage. The debugger is therefore extensible and provides user-programmable breakpoints and the actions to be taken when a breakpoint is encountered are also user-programmable.

## 3. Interactive programming

For programming the Warp, we have designed a language called W2 and implemented an optimizing compiler. The programming model supported by the language hides the low-level details; the programmer sees Warp as a linear array of sequential processors.

This programming model requires that the user partition the computation and data suitably onto the cells in the array. Input partitioning, output partitioning, and pipelining are three useful strategies [2]. For the problem domain of neighborhood operators in computer vision, the Apply system allows the programming of Warp by mapping high-level descriptions that are machine-independent into the W2 language [10]. The problem of devising an efficient algorithm for Warp is interesting in its own right but for the context of this paper, we present those steps that have to be taken *after* a Warp (W2) program has been written. That is, we are mainly interested in the execution aspects.

### 3.1. Program execution

The Warp shell models the Warp array as an integrated part of the user's execution environment. The Warp array operates on shell-level variables that can be initialized, modified, or tested by the user. For example, to create a Warp shell variable, the user enters a var command:

```
var -name RESULT -type INTEGER
```

This first example declares a variable RESULT of type INTEGER. The type of a variable is not limited to the built-in types (integer, floating point, byte, and short (16bit) integer), but the programmer can define new types on the fly

type IMAGE array[512 512] of byte  
This command defines a commonly used type for image processing.

```
var -name IN -type IMAGE  
    -init -textfile /usr/img/road.mip  
var -name OUT -type IMAGE
```

The second example shows how a textfile is used to initialize a variable of type IMAGE. These variables can be displayed (more in Section 3.1.2), or they can be parameters for routines executing on Warp. For example,

`/usr/web/egsb1/egsb1` is a compiled Warp program that takes as input a image of 512×512 bytes and an option (of type integer) and deposits the result in another variable of the type `IMAGE`. The two Warp shell variables `IN` and `OUT` can hold the parameters for an execution of `egsb1`. The command

```
execute -file /usr/web/egsb1/egsb1
        -parameters IN 1 OUT
```

will then run the `egsb1` program. (All command names and keywords can be abbreviated but for sake of clarity this has not been done in this paper. Also, all command names can be redefined by the users to suit individual tastes.)

These three simple shell commands (`var`, `type`, `execute`) capture the basics of executing user programs on Warp. Although they have a simple, intuitive meaning, the actual operation is more complex. Definition of a variable allocates space in the user's shell environment. This variable resides locally in the shell process on the user's workstation. If the file is initialized, then the content of the appropriate file is read.

The `var` commands set the stage for program execution. When the `execute` command is encountered the first time, the shell checks the user's profile for the preferred Warp machine (unless the user has given explicit directions to choose a different Warp machine). Then the shell negotiates with the Warp server on the Warp host to spawn a user process on the Warp host. This process, called the user server, caches the values of variables defined in the user's local shell, and since this process is just started, its state is updated. The values of all shell variables are transferred to this process. After this is done, the user server is queued for access to the Warp array. The Warp monitor guards access to the Warp array and maintains a queue of processes that want to use the array.

After all prior requests for Warp usage have been satisfied (or withdrawn), the Warp array is allocated to this user to run the `egsb1` program. At this time, the user's variables are copied once more: they are moved from the user server's address space to the cluster processors for transfer to the Warp machine. Then the program is downloaded and executed. When the program completes (or times out, or is aborted), the user variables (including any results) are copied back into the address space of the user server, and the Warp array is released.

### 3.1.1. Multiple states and transfers

The user server represents the connection between the user's shell on the workstation and the Warp machine. The above protocol was designed with the goal in mind to tie up the Warp machine as little as possible. Therefore, the long transfers over the slow network occur before the Warp array has been reserved. The transfer from the user server to the cluster processor memories is local to the Warp host; it takes little time compared to the first transfer.

If a user executes a sequence of `execute` commands, the programming environment will optimize the transfers between the user's workstation and the Warp host. Since the user server on the Warp host implements a cache of the user's variables, they need to be transferred only if there were changes. Also, if the same program is executed repeatedly, then there is no need to download the code each time to the system. The Warp monitor checks the state of the instruction memories and optimizes unneeded transfers. (Microcode loaded for program A can be reused even if another program B was executed after the last run of A, as long as the code for B did not destroy the code for A. The monitor overwrites old microcode only if there is no other space available.)

Even when the user executes a sequence of `execute` commands in a row, there is no guarantee of uninterrupted execution. Another user can always sneak in. For those instances that require repeatable timings (or for demonstrations), the user can lock the Warp array, guaranteeing exclusive access.

If for some reason the Warp host crashes, all user servers die. If the user attempts to use Warp again, and the host has been rebooted, the Warp shell detects that the connection to the Warp host is no longer valid. In this case, the session server starts a new process and initializes the state of this server based on the state of the user's shell process on his workstation. This feature proved extremely useful and increased user acceptance of the programming environment significantly. We also discovered an un-intended use: for about a week the Warp host system was unstable while we integrated new processor boards into the system (the boards were delivered "debugged" but when we inserted them, new and unknown problems surfaced). During this time, the system would crash once or twice per hour, but the Warp shell masked these failures for the users, allowing them to debug their code. Also, once the programming environment supported this resiliency, the support of multiple Warp machines was trivial. If a user wants to switch from one Warp to another, the system simply kills the user server on the old Warp host and directs the next request to the new Warp host.

### 3.1.2. Display

The integration of command shell and Warp runtime system offers the opportunity to simplify inspection of Warp variables. Since a large number of Warp users work with images, the Warp programming environment includes commands customized for this application area and interfaces to the Generalized Image Library [11]. The Warp shell command

```
show -var IN
```

is the most basic form to inspect a variable. A new editor buffer is opened and the variable is shown in the buffer. If an array is to be displayed, the user can select individual elements or slices, and arrays are printed in row-major order. Modifying the buffer and writing it out modifies the corresponding variable. Since most programmers have a personal workstation, we support the display of standard 512×512 images on the workstation console. The command

```
show -var OUT -display window1
```

displays a halftoned (greyscaled) image where `window1` is a pre-defined window of the X window manager system. When the command is issued, the system checks if the Warp array is free and uses the Warp array to halftone the image; then the improved image is shipped over the network to the workstation. If the Warp machine is not available, the original image is sent, and greyscaling is done on the workstation (at much lower speed). Figure 3-1 shows a sample user screen snapshot of a typical WPE session.

The user has defined two Warp shell variables `IN` and `OUT`, and a Warp shell command file has been executed that filters a road image stored in `IN` and writes the result into `OUT`. This command file (shown in the lower third of the editor window) calls upon a routine of the `WEB` library, a collection of vision programs implemented on Warp [12]. The input road is shown in the upper left X window, and the filtered road is displayed in the lower right X window.

### 3.2. Debugging

The debugger allows the user to set source line breakpoints and to inspect the state of the cells on symbolic level. A user sets breakpoints by executing the `break` command at the shell level.

**Figure 3-1:** Screen layout

```
break -line 43 -file test2.w2
break -line 123 -file test3.w2
      -cell 0 1 2 8 9
```

The `break` command permits the user to set a breakpoint in the current W2 program. Breakpoints consist of four parts: status, cell, condition and action part. The status specifies whether a breakpoint is enabled or disabled. The cell part specifies the Warp cells to which the breakpoint applies. The condition part specifies a boolean predicate. If it evaluates to TRUE the action part will be executed.

The default condition for a breakpoint is the TRUE predicate, which always evaluates to true. That is, whenever execution reaches the specified line, then the breakpoint is taken. The default action is `halt`, which stops execution of the Warp array. The user then inspects the state

```
get temp -cells 1
get -locals -mode hex
get -locals -function init
get -globals
```

If the `-cells` option is specified, the values are shown for the selected cells, otherwise only the values of the cell in which the current breakpoint occurred are shown. The `-mode` option permits the user to specify one of several display formats.

In those cases when the default action is not appropriate, the programmer can specify a different condition and action part by setting the environment variables `BREAKACTION` and `BREAKCONDITION`, respectively.

Breakpoint conditions are Common Lisp predicates, and it is possible to mix Lisp and W2 values in breakpoint conditions. W2 values can be retrieved by calling the Lisp function `get` with any argument that is a valid argument to the Warp shell command `get`. `get` returns W2 values in the internal representation used by Common Lisp. Examples of breakpoint conditions are:

```
(= (get "i") 1)

(= (get "--value global") 1.22333)

(and (= (get "--value i -cell 1") 2
        (= (get "--value i -cell 2") 3)))

(= (get "--value array") #(0.1 0.2 0.3))
```

For example, the first predicate retrieves the value of a variable (`i`) from the array and compares it with the (Lisp) constant "1". These conditions allow the user to custom-tailor the debugger; implementation details as well as performance results are presented in a separate paper [6].

## 4. Real-time applications

The previous section illustrated the programming environment for interactive users. Real-time applications often do not need or want to execute commands from the Warp shell.

For those users, the Warp monitor also supports applications which do not need the full functionality of the Warp shell but must run as fast as possible. In this case programs can run in standalone mode and call the Warp monitor functions explicitly. This mode is supported both for remote execution over the network and for execution locally on the Warp host. This mode is the mode with the lowest overhead and is the preferred mode of execution when time is critical. Application programs in standalone mode can be written in any language as long as the language implementation supports the call of external C routines (the Warp monitor is written in C).

### 4.1. Execution

The invocation and execution outside of the shell is straightforward from a user's standpoint. Using the Warp User Package, the interface from a C programs looks like this:

```
#include <stdio.h>
#include "monitor.h"

/* call W2 program for
   computing edges in input image */

if (wu_call("/usr/web/egsbl/egsbl",
           IN, 1, OUT) != 0)
    wu_error();
```

The system actions are the same as if the user had invoked a `execute` command with `IN` and `1` as input parameter and `OUT` as output parameter. The code segment shown above captures completely the steps necessary to transform the interactive call from Figure 3-1 to a call in a standalone, batch program.

### 4.2. Migration

While it is important to support the execution of programs from other environments than the Warp shell, the real problem in supporting real-time applications is not in providing a method for C (or FORTRAN programs) to invoke programs on the Warp machine. The much harder problem is to provide a good environment to develop such applications. The use of the interactive environment is beneficial if there exist an easy and consistent migration path from the interactive environment to the real-time environment. Unless the path from the interactive environment to the real-time environment is trouble-free, the real-time programmer is not helped at all, since any problems that occur during the conversion have to be debugged under the constraints of the real-time environment.

We addressed this issue by implementing the Warp monitor in such a way that there is no difference between the core image (object code file) of an application running in remote mode or in local mode. Thus application programmers can

compile, link and test their applications in the familiar environment of their personal workstation before they download them to the Warp host for local execution. In our environment, programmers will test individual programs in the interactive mode and then combine a number of Warp routines into a module to solve an application problem.

This mechanism addresses the issue of moving applications from the interactive environment to the real-time environment. However, this does not handle the frequent situation that the programmer discovers some problem in his code after the system has been integrated. At this time, debugging the offending program in isolation is often useless since the input to the Warp array might be computed or modified by a previous phase of the application module. For this case, the Warp monitor provides the option to build a "scaffold": when the `wu_call` is issued, the values of all input parameters are written to disk, and enough information about the types of the variables is kept to allow replay of this step interactively at the Warp shell level.

## 5. Measuring WPE overhead

The system outlined in the previous sections provides the desired functionality, but we also have to demonstrate that the performance requirements are met as well. We compare the total elapsed time with the computation time on the Warp array to determine the overhead introduced by our programming environment and runtime system. The overhead experienced by a WPE user depends on the particular usage mode. To illustrate the full range of possible overheads, we define the following usage modes which characterize significant stages in the life cycle of a Warp program.

In the *interactive mode*, the Warp machine is accessed remotely via the Warp shell. This mode is typical for the developer of a W2 program; the various WShell processes in Figure 1-1 represent example users. In the *remote batch* mode the Warp machine is accessed remotely via the Warp monitor which is typical for application programmers who want to test the application from their workstations without real-time requirements. Direct 1 of Workstation 2 in Figure 1-1 is such a user. The *local batch* mode, the mode of choice for time-critical applications, accesses the Warp machine locally using the Warp monitor, like Direct 2 in Figure 1-1. We also distinguish whether the Warp machine is already acquired by the user or not. In *single shot* mode, Warp is not yet locked, whereas in *multiple shot* mode, Warp is already locked before the execution starts. Finally we distinguish whether a W2 program is executed the *first time* or *repeatedly*.

For the measurements we selected a sample of five W2 programs. A program called EMPTY consisting of a single empty statement is included in this sample to illustrate the *pure* overhead caused by the various usage modes. The other programs in the sample are real W2 programs taken from the WEB library [12] and show the actual overhead experienced by a WPE user. FFT performs a fast Fourier transformation, HIST computes the grey value distribution of a 512×512 byte image, SUMRCR sums the rows and columns of an image, and THRESH enhances the contrast of a 512×512 byte image.

To make the execution times of each program comparable, we make several assumptions. First, for remote access, we assume that the user server is already forked off (on the average this takes 1.8 sec). Second, we assume that user

**Table 5-1:** Execution times of five W2 programs

server memory for input and output variables is already allocated (780 - 1100 msec). The last assumption is that all input variables are already initialized (1140 msec per image).

For each possible combination of execution, locking and repetition mode just introduced we executed the sample multiple times. The results of the experiments are summarized in Table 5-1. For each of the five programs, we provide data for the different usage modes. The last line of Table 5-1 gives the execution time on the Warp array.

The least overhead is encountered for *repeated multiple shot batch local mode* (MSr-bL). That is, a standalone program running on the Warp host executes repeatedly the same function on the Warp array. In this case, the execution overhead caused by WPE is only 5-6 msec (difference between execution time on array [last line] and line labeled "bL" for "Multiple Shot (repeat)" execution). The total elapsed time reported includes reading the input variables from cluster memory into the Warp cells, executing the W2 program and writing the results back to cluster memory. In the Warp machine, I/O from the host to the array is overlapped with the execution of the program: As soon as the first byte of the input variable is available in the first cell, the execution can start. Reading or writing a 512x512 byte image, that is, transferring the image between cluster memory and Warp array, takes only 30 msec because the Warp machine supports DMA to the array with a peak bandwidth of about 8 Mbyte/sec. As we can see, the execution overhead is the same for all W2 programs in the sample, and none of the programs is I/O bound.

The additional overhead caused by *first multiple shot batch local mode* (MSf-bL) is 116 to 178 msec. This is the time it takes to copy the cluster code and cell microcode into the cluster processor memory and Warp cell program memory, respectively. The copy is always done when a W2 program is executed after the Warp machine has been locked.

The additional overhead caused by *repeated single shot batch local mode* (SSr-bL) is 4011 to 5604 msec. This is the time it takes to lock the Warp machine before the execution starts until it is unlocked afterwards. Locking the Warp machine involves the copying of all input variables from user server memory in the Warp host to cluster memory; unlocking the Warp requires copying the variables from cluster memory back to user server memory.

The additional overhead caused by the *first single shot batch local mode* (SSf-bL) is 675 to 8930 msec. This is the time it takes to read the symbol table, cluster code and microcode files into user server memory. All our experiments are performed on diskless SUN-3 workstations, and each file lookup is done with SUN's NFS system. This explains fluctuations in the local and remote batch execution times in single shot mode. For example, it took longer to execute FFT locally than remotely (18204 msec vs 17048 msec). This

overhead can be expected to be much less if local file lookup is used.

The additional performance penalty incurred when accessing the Warp machine *interactively* is in the range of 2-8 seconds. We believe that this overhead is tolerable, especially during the development cycle of a W2 program.

## 6. Concluding remarks

The Warp Programming Environment has been in use now for more than a year. Its versatility as a development as well as an execution environment and a consistent migration path between these environments makes it attractive for many people. At Carnegie Mellon University, WPE is currently used by about 30 people for implementation and development of the environment itself and for the implementation of Warp programs. Several Warp machines have been delivered by GE, Carnegie Mellon's industrial partner for this project, and these systems run the WPE system. It is currently being used as the base for computer vision research, for image processing, for neural net simulations, signal processing applications, and numerical analysis. One Warp machine is mounted in a robot vehicle and is used by the computer vision group in its NABLAB project mainly for real-time applications.

The integration of the program development environment with the execution environment has made the Warp machine much more accessible to its users. The essential part of the runtime environment needed for real-time applications is a complete, well-defined subset of the overall programming environment. By taking care to clearly define the interfaces between the programming environment and this runtime system, we allow the user to easily move from the interactive development stage to real-time execution and back. At the same time we have a system that exhibits a low overhead for those real-time applications when running locally on the Warp host. Supporting these two different usage modes by an integrated environment has reduced and simplified the work required to implement the environment. At the same time, the user is offered a convenient migration path from program development to real-time execution.

## Acknowledgments

The Warp shell is based on the Lisp shell developed by Dario Giuse. The W2 compiler was developed by the compiler group that included Chang-Hsin Chang, Robert Cohn, Monica Lam, Peter Lieu, Abu Noaman and David Yam. The W2 simulator was implemented by Angelika Zobel. The Warp monitor was implemented by Michael Browne. Ed Clune suggested several extensions to the Warp monitor. Parts of the Warp monitor are based on an earlier runtime environment called WARPCI developed by Francois Bitz and Jon Webb. The Generalized Image Library was developed by Leonard Hamey. Marco Annaratone, Robert Cohn, Harry Printz and Leonard Hamey were the first users and made valuable suggestions.

## References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., Sarocky, K., and Webb, J. A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, IEEE/ACM, June, 1986, pp. 346 - 356.
2. Annaratone, M., Bitz, F., Clune E., Kung H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J. Applications and Algorithm Partitioning on Warp. Proc. Comcon Spring 87, San Francisco, February, 1987, pp. 272-275.
3. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., and Webb, J. A. "The Warp Machine: Architecture, Implementation and Performance". *IEEE Trans. on Computers C-36*, 12 (Dec. 1987), 1523-1538.
4. Bouknight, W. J. et al. "The Illiac IV System". *Proc. IEEE* (April 1972), 369-379.
5. Bruegge, B. Warp Programming Environment: User Manual. Tech. Rept. CMU-CS-88-105, Carnegie Mellon University, Dept. of Computer Science, 1988.
6. Bruegge, B. and Gross, T. A Program Debugger for a Systolic Array: Design and Implementation. Proc. of the Second Workshop on Parallel and Distributed Debugging, SIGPLAN, Madison, WI, May, 1988.
7. Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K. . A VLIW Architecture for a Trace Scheduling Compiler. Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, CA., Oct., 1987, pp. 180-192.
8. Dunlay, T. R. Obstacle Avoidance Perception Processing for the Autonomous Land Vehicle. IEEE Intl. Conf. on Robotics and Automation, Vol. 2, April, 1988, pp. 912 - 918.
9. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.
10. Hamey, L., Webb, J. and Wu, I. Low-Level Vision on Warp and the Apply Programming Model. In Janusz Kowalik, Ed., *Parallel Computation and Computers for AI*, Kluwer Academic Publishers, 1987.
11. Hamey, L. A User's Guide to the Generalized Image Library. In *Warp Programming Environment: Documentation*, Carnegie Mellon University, Dept. of Computer Science, 1988.
12. Kanade, T. and Webb, J. End of Year Report for Parallel Vision Algorithm Design & Implementation. CMU, Robotics Institute, 1987.
13. Lam, M. S. *A Systolic Array Optimizing Compiler*. Ph.D. Th., Carnegie Mellon University, May 1987.
14. Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T. Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. Submitted to the IEEE Second International Conference on Neural Networks, April, 1988.
15. Scheifler, R. and Gettys, J. "The X Window System". *ACM Trans. on Graphics* 5, 2 (April 1986).
16. Stallman, R. *GNU Emacs Manual, 4th edition*. Cambridge, Mass, 1988.
17. Tseng, P. S. Sparse Matrix Computation on Warp. The 3rd International Conference on Supercomputing, May, 1988.
18. Wallace, R., Matsuzaki, K., Goto, T., Crisman, J., Webb, J. and Kanade, T. Progress in Robot Road-Following. IEEE Intern. Conf. on Robotics and Automation, April, 1986, pp. 1615-1621.
19. Young, J. "Supercomputers join the Unix Club". *Electronics* 61, 5 (March 1988).

## Table of Contents

<b>1. Introduction</b>	<b>0</b>
1.1. Warp overview	1
1.2. WPE overview	1
1.3. User classes	2
<b>2. Tools</b>	<b>2</b>
2.1. Monitor	2
2.2. Shell	3
2.3. Debugger	3
<b>3. Interactive programming</b>	<b>3</b>
3.1. Program execution	3
3.1.1. Multiple states and transfers	4
3.1.2. Display	4
3.2. Debugging	5
<b>4. Real-time applications</b>	<b>6</b>
4.1. Execution	6
4.2. Migration	6
<b>5. Measuring WPE overhead</b>	<b>7</b>
<b>6. Concluding remarks</b>	<b>8</b>

## List of Figures

**Figure 1-1: Warp system environment at Carnegie Mellon**

**1**

**Figure 3-1: Screen layout**

**5**

**List of Tables**

**Table 5-1: Execution times of five W2 programs**