The Warp programming environment


Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

This paper describes the environment for developing and executing Warp* programs. The center of the program development environment is a customized shell that ties together a compiler for the Warp array, the Warp run-time system, and a debugger. The compiler translates high-level language programs to microcode for the Warp machine. It achieves a high utilization of the computation power of the processor. The run-time system supports remote execution of Warp programs across a network and makes the Warp machine available as a shareable resource. The debugger permits symbolic debugging of Warp programs. The Warp programming environment makes the Warp machine an easily programmable and accessible attached processor in a UNIX™ environment.

* Warp is a service mark of Carnegie Mellon University.
INTRODUCTION

In our programming environment, Warp is modeled as an attached processor accessible from an interactive, programmable, command interpreter called the Warp shell. The shell provides traditional operating system commands as well as commands to execute programs on the Warp machine. Calling a Warp program is similar to invoking a procedure: the shell calls the Warp program and passes input and output data between the application and Warp. The run-time system provides low-level support such as securing exclusive access to the machine, downloading object code, and transferring data between the host and the Warp system.

For programming the Warp, we have designed a language called W2 and implemented an optimizing compiler. The programming model, as supported by the language, allows the user to see the machine as a linear array of sequential processors and hides the low-level details from users. From a W2 program, the compiler generates microcode for the Warp array and the interface unit, as well as C programs for the I/O processors.

In this paper we first describe the objectives of the Warp programming environment (WPE), and the system configuration. Then we describe the two methods for using the Warp system. The primary method is the interactive mode through the Warp shell; a library of existing Warp routines as well as user programs can be invoked interactively through shell commands. Program development is done almost exclusively with this method. The second method, used mainly for real-time systems, is the direct mode, for users who cannot afford the overhead of an interactive system. We then describe the support software in WPE: the run-time system, compiler, and debugger. We conclude with a review of the current status and a brief discussion of our experience to date.

Objectives of WPE

The primary objective of WPE is to simplify the use of the Warp machine. WPE is a uniform environment to edit, compile, debug, and execute W2 programs. Its audience includes the user who calls routines from a W2 library, the programmer who develops new algorithms for Warp, as well as the implementor who writes support software.

WPE must support efficient multiple user access because the use of the Warp hardware in a typical user session is sporadic. By allowing multiple user sessions to overlap and by serializing the use of the hardware, the Warp machine can be better utilized. WPE also provides multiple machine access; if there is more than one Warp array available, a user has the choice of connecting to any of these machines. It also provides network transparency, the user sees no difference whether he uses the Warp array remotely from his personal workstation or logs in directly to the Warp host machine.

WPE is designed to be development machine-portable. The shell, compiler, and debugger are written in Common LISP, which runs on many workstations, and the TCP/IP protocol is used in inter-machine communication. Our current release of WPE runs on SUN-3 under BSD UNIX 4.2. WPE is also designed to be target machine-portable. It has been in use for our prototype system, and it can be used with the successor Warp architectures: the production architecture implemented with printed circuit boards as well as the VLSI Warp which is currently in the design stage.

System Configuration

Figure 1 shows the configuration of WPE. Each workstation, a SUN-3, runs 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 external host and Warp array.2 The Warp server executes on the Warp host and is the intermediator between users and the Warp array and external host.

TWO MODES OF ACCESSING WARP

There are two methods of running programs on Warp. Users may use the Warp shell which provides an interactive interface to the constituents of WPE such as the compiler, run-time system, debugger, and servers. Or, if absolute performance is necessary, users may program the machine in direct mode, without the overhead of a command interpreter.

The Warp Shell

The Warp shell binds together the components of WPE. Shell commands can be used to invoke the compiler, run a program on the array, and call debugging functions. The Warp shell is based on an extensible shell written in Common LISP.3 The extensibility makes it possible to support different classes of users. Specifically, the Warp shell distinguishes between the novice and the experienced user. For example, the implementation language Common LISP and the components of the environment are completely hidden from a novice. This is useful for programmers interested in using the Warp shell to execute W2 programs from a library. On the other hand, the LISP implementation and all the software components comprising the Warp environment are easily available when de-
sired. This means an experienced user can make use of Common LISP's powerful control structures to implement new commands.

The Warp shell allows the declaration and manipulation of variables, which can be used as inputs or outputs for Warp programs. All variables in the Warp shell are typed. The type information determines how to present a variable to a user (print as integer, floating-point number, ...) and how to transfer it to the Warp array. The Warp shell offers predefined and user-defined types and variables. For example, an image can be defined by a user as a type "IMAGE = array [512,512] of byte" and the user can define variables of type IMAGE. User defined variables can then be passed as parameters to W2 programs.

Let us assume the user wants to invoke a W2 program "filter5by5" contained in a W2 library. This program expects an input image and transforms it into an output image. A typical sequence of Warp shell commands looks like this:

allocate -name IN -type IMAGE -init /img/road
allocate -name OUT -type IMAGE
filter5by5 IN OUT

The first command defines a Warp shell variable IN of type IMAGE and initializes it with the data contained in file "/*img/road." The second command defines the variable OUT to hold the output image. The third line invokes the W2 program "filter5by5" with the actual parameters IN and OUT. When the execution is finished, the output image can be displayed or inspected in an editor buffer.

The user types commands to the Warp shell which runs inside the editor. The advantage of this structure is that such features as intra-line editing, history buffers, re-execution of previous commands and creation of script files are available automatically. The Warp shell also provides a uniform help mechanism. Each command is documented on-line and examples from the help description of a command can be fed to the command interpreter, providing easy exploration of the command language.

In addition to the Warp-specific features described in this paper, the Warp shell provides roughly the functionality of the well-known UNIX C-shell. It maintains a set of environment variables such as SOURCEFILE, the name of the W2 program; HOST, the name of the Warp host in use; and BREAKPOINTS, the set of currently defined breakpoints. These environment variables can be inspected and assigned new values with Warp shell commands. By setting variables, a user can configure the environment. For example, assigning a value to HOST changes the Warp host and array on which programs are executed.
**Direct Mode**

The Warp shell is programmed in Common LISP and therefore garbage collection occurs regularly, making it hard to achieve predictable response times at the shell level. In addition, there is some overhead incurred in the network communication. Although this is tolerable when developing Warp programs, it may not be acceptable for real-time applications. In this case, a user calls the run-time system directly. Application programs in direct mode can be written in any language. The only requirement is that the language implementation supports the call of external C routines (the run-time system is written in C).

Direct mode is supported for both remote and local execution. Applications running remotely still use TCP/IP; application programs executed locally on the Warp host bypass the TCP/IP protocol. In Figure 1, “Direct 2” and “Direct 1” are examples of the remote and local direct mode, respectively.

The local direct mode is the mode with the lowest overhead and is the preferred mode of execution when time is critical. In this mode, the application program makes direct procedure calls to the run-time system. A program in this mode can run only on the Warp host because it is linked with the library into a single UNIX process.

**SUPPORT SOFTWARE**

Figure 2 shows the major software components of WPE; a compiler, a debugger, an editor, the Warp server, and the shell user interface. The different components of the environment communicate via the WPE database, which contains the W2 source files, symbol tables, and syntax trees. The shell's environment variables capture the current state of the session, for example, which Warp machine is allocated and what class of user (level of experience) is using the system. The integration of the compiler's internal tables with the shell and the debugger is important for the functionality of WPE. For example, the syntax tree produced by the W2 compiler is accessible by other components of WPE. The debugger inspects the syntax tree when a user tries to set a breakpoint. When execution on the cell reaches a breakpoint, the corresponding line is displayed in an editor buffer. Another part of the database that is used frequently is the symbol table. The Warp shell examines the symbol table when it displays the value of a variable or when it matches the actual parameters of a Warp program call with the formal parameters of the program.

**The Run-time System**

The WPE run-time system supports multiple user access by including two kinds of servers, the Warp server and the user
servers. These server processes run on the Warp host and communicate with the Warp shells on a user’s workstation via remote procedure calls using the TCP-IP protocol.

For each user running either the Warp shell or in direct mode on a remote machine, there is a process called the user server which also resides in the Warp host. Variables created by a user in the Warp shell reside physically in the user server’s address space on the Warp host. Variables are transferred to the user’s site only when necessary. This organization reduces the performance impact of running the Warp shell remotely and accessing Warp over the network. Furthermore, because Warp shell variables are allocated in the user server, they can be initialized without possessing the Warp array. This makes it possible to perform costly file transfers (like reading an image) between the workstation and the Warp host before locking the Warp machine.

The Warp server manages the access to the Warp machine. It provides functions to lock and unlock the Warp. When the Warp server locks the machine for a particular Warp shell, the memory of its user server is copied into the memory of the external host of the Warp machine. This copy operation is done inside the Warp host and is therefore quite fast. When the Warp machine is unlocked, the memory of the external host is copied back into the memory of the corresponding user server. This scheme permits the environment to maintain user-specific state information across several locks/unlocks of the Warp machine.

The run-time system provides for sharing the Warp machine, but does not preempt a user once the Warp machine is locked. Most programs require a few seconds to run; the overhead associated with swapping processes is too high compared with the execution time.

The W2 Programming Language and Compiler

In Warp, parallelism exists at several levels. At the cell level there is a horizontal architecture with multiple pipelined functional units; at the array level there are ten cells; at the system level there are separate processors in the external host for input/output, control, and computation. The potential performance of Warp is enormous, but the complexity of using the machine is proportionally overwhelming. To harness the computation power of Warp, we have designed a programming language called W2 and implemented an optimizing compiler. The W2 language provides an abstract programming model of the machine that allows users to focus on the parallelism at the array level. The compiler handles the parallelism at the system and the cell levels.

Programming model

Users view the Warp system as a linear array of identical, conventional processors that can communicate asynchronously with their left and right neighbors. Standard language constructs such as loops and conditionals are provided, as are primitives for sending and receiving data. The semantics of the communication primitives are that a cell will block if it tries to receive from an empty queue or send to a full one.

The general problem of partitioning a computation for a processor array is difficult to solve. Usually, a solid understanding of the application domain is necessary to find a good mapping of a computation onto a processor array. Therefore, the processor array configuration is exposed in the programmer’s model, giving the user or higher-level tools full control over computation partitioning. Already there are application-specific tools that map sequential algorithm descriptions into parallel W2 programs.

The W2 programming language

The W2 language is a simple block-structured language with assignment, conditional, and loop statements. A W2 program is a module; it defines the interface between the host and the array—the input and output to and from the array are given by the module parameters. Specified next are the cell programs, each of which describes the action of a group of one or more cells. Only one cell program is allowed for the prototype machine. When a group of cells share the same program, it does not mean they necessarily execute the same instruction at the same time. In fact, computations on different cells typically are skewed in a pipelined fashion because a cell cannot start executing until it receives data from the preceding cell. Finally, a cell program may consist of several unnested functions.

Example program

Figure 3 is a simple example of a Warp program which evaluates a polynomial using an array of ten cells. The program evaluates the polynomial

\[
P(z) = c_0 z^9 + c_1 z^8 + \cdots + c_9 = (((c_0 \times z) + c_1) \times z + \cdots + c_8) \times z + c_9
\]

for a vector of 100 input data \(z_0, z_1, z_2, \ldots\) By applying Horner’s rule, a polynomial evaluation becomes a series of inner-product computations, each of which is computed on a cell in the array. Each cell (starting with cell 0 up to cell 9, the last cell in the system) executes a copy of the program. The first cell receives the values of the host program variables (bound to parameters \(c_0\) and \(z\)), and the results are sent and stored in a host variable bound to parameter “results.”

The compiler

The local optimizations implemented include common subexpression elimination, constant folding, height reduction, dead code removal, and idempotent operation removal. A global flow analyzer collects detailed inter-block information for all variables of the program. For regular accessing patterns, the analysis is powerful enough to distinguish between individual array elements and different iterations of a loop so that the code generator can overlap different loop iterations. To exploit the high degree of pipelining and parallelism in the machine, the compiler has a good global scheduler. We use
module polynomial (z in, c in, results out)
float z[100], c[10], results[100];
cellprogram (cid : 0 : 9)
begin
function poly
begin
float coeff, /* local copy of c[cid] */
temp, xin, yin, ans; /* temporaries */
int i;
receive (L, X, coeff, c[0]);
for i := 1 to 9 do begin
receive (L, X, temp, c[i]);
send (R, X, temp);
end;
send (R, X, 0.0);
for i := 0 to 99 do begin
receive (L, X, xin, z[i]);
receive (L, Y, yin, 0.0);
send (R, X, xin);
an := coeff + yin*xin;
send (R, Y, ans, results[i]);
end;
call poly;
end
end

Figure 3—Example program

two scheduling algorithms: a scheduling technique specialized for innermost loops called software pipelining, and a new unified approach to scheduling both within and across basic blocks.\(^7\)

The Debugger

The Warp debugger provides two functions: setting source line breakpoints and symbolic inspection of variables. Because the optimizing compiler deletes redundant operations and reorders source operations, it is not always possible to set a breakpoint at a particular line in the W2 source code. A special Warp shell command permits a user to explore possible breakpoints. For a machine that executes 100 million operations per second, a simple line-oriented debugging model is not always appropriate. We must be able to qualify the breakpoint with a condition so that the program automatically resumes execution at the breakpoint if the condition is not satisfied. For example, we need to be able to stop at some particular iteration of a loop, without stopping at all the previous iterations.

For the wire-wrapped prototype, we can provide only post-mortem debugging; insufficient access to the internals of the cell makes it impossible to continue execution after resources have been inspected. This problem is alleviated in the production version of the Warp array.

CURRENT STATE

WPE is implemented in Common LISP and C, and is running under BSD UNIX 4.2 on a SUN-3 Workstation. The current release supports multiple users and multiple machine access to two copies of the 10-cell wire-wrapped prototype.\(^8\) The core image of Common LISP is about 7 MBytes; WPE uses an additional 3 MBytes. We have found that a paging space of 25MBytes per user provides acceptable performance.

Altogether, approximately 75,000 lines of code have been written. The W2 compiler accounts for about 34,000 lines of Common LISP code, and the assemblers for 16,000 lines of C code. The shell contains 8,000 lines of LISP code; it relies on a text editor (Emacs). The debugger contains about 3,000 lines of LISP code. The run-time system is written in C and consists of 4,000 lines of code. Linkers and simulators account for the remaining lines of code.

Table I presents the performance results for some well-known programs for the prototype Warp system. The second column shows the maximum floating-point computation bandwidth that can be obtained for each program. Since there are two distinct functional units for addition and multiplication, this maximum rate is less than 100 MFLOPS if the number of additions is not equal to the number of multiplications. These numbers do not take into consideration the data dependencies in the program, but only the total number of operations. The next column presents the computation bandwidth achieved by the microcode generated by the compiler. The overhead incurred by the host is not included.

Development of the Warp programming environment started in 1984 as the architecture was defined. The major emphasis of the early work was on the programming language definition and the compiler design. Since the first prototype machine became operational in Spring 1986, increased effort has been allocated to the run-time support and the user interface. The programming environment has been continuously developed and improved, with input from our application users.

WPE provides a uniform environment for developing and running Warp routines. The massive amount of details in using the machine are abstracted out; efficient run-time support

<table>
<thead>
<tr>
<th>Program</th>
<th>MFLOPS max</th>
<th>MFLOPS actual</th>
<th>Execution (ms)</th>
<th>Compilation (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution (3x3 kernel)</td>
<td>94.4</td>
<td>65.3</td>
<td>68</td>
<td>4.9</td>
</tr>
<tr>
<td>Matrix multiply (100x100)</td>
<td>99.5</td>
<td>74.5</td>
<td>25</td>
<td>1.7</td>
</tr>
<tr>
<td>Successive over-relaxation (225x225, 10 iterations)</td>
<td>88.9</td>
<td>45.0</td>
<td>180</td>
<td>2.6</td>
</tr>
<tr>
<td>Local average selective filter (512x512 image)</td>
<td>65.3</td>
<td>42.2</td>
<td>396</td>
<td>9.7</td>
</tr>
<tr>
<td>Mandelbrot (512x512 image, 256 iterations)</td>
<td>90.0</td>
<td>86.8</td>
<td>6960</td>
<td>5.0</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
is easily accessible through an interactive command interpreter. The run-time system also allows multiple user access and greatly increases the utilization of the hardware. The development of software for Warp is made easy by a highly optimizing compiler, which generates efficient microcode from a high-level language. Microprogramming has been phased out completely since the compiler became functional. In summary, the Warp programming environment has turned the Warp machine into an easily programmable and accessible attached processor in a UNIX environment.

ACKNOWLEDGEMENTS

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. T. Gross is also supported by an IBM Faculty Development Award.

REFERENCES