External Product Specification for the iWarp Programmed Communication Services (PCS) Array Combiner

Thomas Gross and Susan Hinrichs
August 1991
Version 2.0

Abstract

The combiner provides a way to link independent single-cell object modules to form an array module.

This document describes *what* functionality is provided by the iWarp combiner, at a level that is suitable for a member of the iWarp software team. A later document describes *how* the iWarp combiner works.

1 Introduction

This document is the External Product Specification (EPS) for the iWarp PCS combiner **iwcb**. This document describes the general behavior of the combiner, specifies how the combiner interacts with the environment of the machine it is executed on, and how the user interacts with the combiner.

An iWarp system consists of one or more iWarp cells; each iWarp cell consists of an iWarp component and some local memory. The iWarp component provides sophisticated support for inter-processor (cell-to-cell) communication. We assume the iWarp system is arranged in a regular $m \times n$ torus.

The combiner executes as part of the program development tool chain. It is executed on a host system (i.e. it is a cross-combiner). The iWarp combiner has no equivalent in a standard microprocessor environment, which is usually uni-processor oriented. The iWarp combiner is developed exclusively for the iWarp system.

1.1 General combiner functionality

An iWarp application is characterized by

- the object code (program) that executes on each of the iWarp cells
- the connections between these cells that are used by the programs for inter-processor communication (*optional*).

The physical connections (i.e. the physical busses that connect two iWarp cells) of an iWarp system are fixed, but the communication system of the iWarp cells provides the option to configure connections between different cells in the system. These connections can span over multiple cells and are application program specific (that is, different programs use different connection setups). The number and routing of these connections are limited only by the total number of communication resources available on each iWarp cell. (The number of communication resources available is a function of the runtime system as well as of the iWarp component hardware). Applications that include such connections for communication use the Program Communication Service (PCS), which is described in a separate EPS, to initiate and manage the actual transfer of data. These applications use the combiner to describe the connection set up and the object code for each cell. Applications that do not include such connections for communication can still use the message passing services like those offered by Express. The programmer can also reserve connection resources from the combiner to create dynamic connections at runtime.

The iWarp combiner **iwcb** combines the object code for multiple cells of an iWarp array, together with a description of the connections between these cells, to generate an array module. The output of the combiner is an array module that is input to the iWarp array loader for execution (activation) on an iWarp system. The array module contains copies of the single cell programs and cell mapping information, so the individual cells can be loaded by the iWarp array loader. The array module also contains descriptions of the program-visible connections between the cells. Connections between cells defined in the array module are *static*; i.e. they are established and known before execution starts and persist for the whole duration of execution.

The input to the combiner is always specified for a logical array with cell (0,0) as *origin*. The input to the array combiner is an array program that specifies a mapping between cells programs (produced by the single-cell tool chain) and individual cells and can include previously defined array modules. The array program also describes the communication connections between the various cells and reserves resources for connectections either implicitly for static connections or explicitly for dynamic connections. There may be no static communication connection descriptions if the program relies on messages passing or other runtime connections. The combiner checks that the given connections are legal and issues an error message if it detects an example of illegal use. The array program specifies the cell mapping and program connections by function calls described in section 4.4. These function calls are defined in files included in the array program and libraries linked to the array program by the combiner.

The output of the combiner is an array module, suitable for loading or subsequent combiner steps. The cell programs are included directly in the array module from the single cell tool chain and are not modified by the array combiner. In addition to the cell programs, the array combiner adds the connection and resource reservation information.

1.2 Combiner's place in iWarp tool chain

The program tool chain includes all software tools that aid in the translation and execution of programs for the iWarp system. There are five distinct steps that have to be taken to transform a problem specification (of an application problem) into a set of object codes executing on an iWarp system.

We assume the problem to be mapped onto an iWarp system is initially specified without reference to the specifics of the target array. That is, the problem assumes a single thread of control and a single data space; there are no explicit communication operations. From this, a set of programs, one for each cell, must be generated. At the time these individual programs are generated, the computation and the data are distributed over multiple cells, and appropriate communication statements for data exchange and synchronization must be created. Such translation can be done by a *parallel program generator*, a program that produces a parallel program for execution on the target array, or the translation can be done by a programmer. Parallel program generators are important to make the programming of parallel systems practical, and several parallel program generators have been build at Carnegie Mellon, and one of them (Apply) has been retargetted and ported by Intel Corp. to iWarp. A program for a single cell is called a *cell program* and is written in C, a language that has an iWarp cell compiler.

Cell programs use *ports* to identify connections between different cell programs. A port is a local name, in a single cell program, that refers either to a incoming or outgoing pathway, as depicted by Figure 1. In a later step, the information about port connectivity (port `xout` of cell (i,j) is connected to port `xin` of cell (i, j+3)) is used to determine which pathways are needed for this application. Cell programs can use the PCS data movement library to access the communication networks through these ports.

The second step of translating a collection of cell programs for execution on an iWarp array is *compiling*. Each cell program (for a single cell of the logical array) is compiled with a suitable compiler. Note that this compiler is a 'single-cell' compiler; that is, the compiler has no knowledge of the final placement of the cell program in the array. This compiler does not deal with the interactions of one cell program with another cell program in the array. A side effect of this model of translation is that the same object code (produced by the local compilation) can be used in various scenarios, e.g. as part of a 3×3 as well as a 4×4 application, provided that the same cell program can be used. This is quite useful when creating libraries, etc.

The third step is *linking* the compiled cell programs. For each cell program, unresolved names to external objects (functions, variables) are taken care of. Note that cell programs can only refer to objects on the same cell, so this step is no different than the linking step usually included in compiling for a

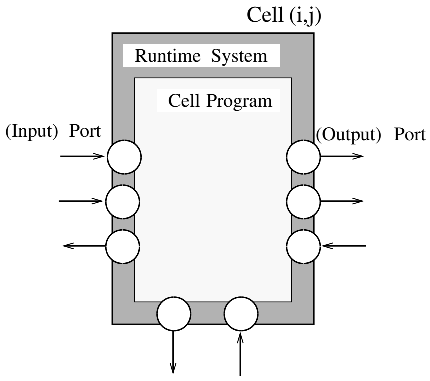

Figure 1: Single-cell program with ports

conventional uniprocessor. The linker takes care only of intra-cell references. As in any other tool chain, the linker can be invoked multiple times.

The iWarp communication agent provides for a variety of connections and communication paths between individual cells. The set-up of these connections requires the cooperation of all cells in the array. So the next step is to produce the initialization code from the user's description of the communication paths in the machine. This step is called *array combining* because at this time the individual cell programs (which so far are unrelated) are combined into an array module. The functionality for this step is provided by the iWarp combiner **iwcb**, which is discussed in this document.

Figure 2 shows a simple example of three cell programs that have been combined. Notice that there is a non-neighbor communication connection between the first and third cell.

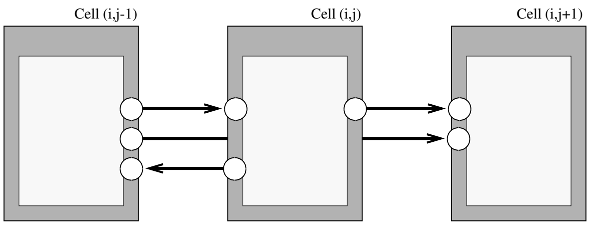

Figure 2: Multiple cell programs combined

Multiple cell programs can be combined to create an *iWarp array module*. In practice, it makes no difference if the cell programs that are combined into a module have been written 'by hand' or have been generated by a parallel program generator; they are combined the same way. The only difference is that if a parallel program generator is used, the parallel program generator will produce the input for the combiner as well.

The last step is process *activation*. This step includes the assignment (placement) of cell object code onto physical iWarp cells, relocation of the object code (if needed), and any linkage with the runtime system that is necessary to provide input/output capabilities to the application executing on the cells, as well as the downloading of the program codes (and initialization of memory, communication agent etc. as appropriate). At that time, the program is ready to execute. The iWarp array loader takes care of process activation.

Figure 3 illustrates the sequencing of the various translation steps that turn a collection of cell programs into an array module that can be executed on an iWarp array.

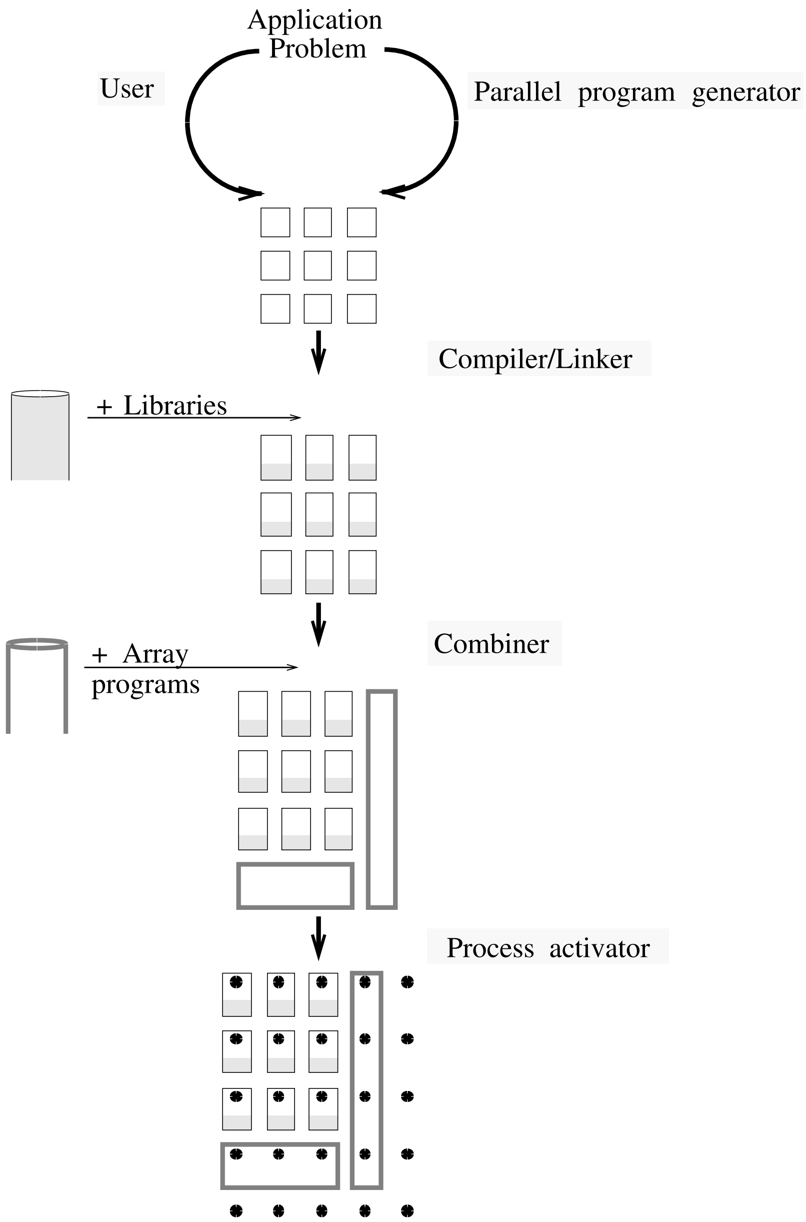

Figure 3: Program translation steps

1.3 Scope, purpose, and background of this document

This combiner EPS does not repeat any information about the single-cell tool chain. Information about these tools can be found in the respective EPS of each tool.

The combiner assumes a static view of the iWarp array. That is, during the execution of a program on the array, the number of cells, the properties of each cell, and the physical connections between the cells are constant. That is, no re-mapping of an application program takes place during the execution of an application program. Future software tools may address the needs of fault tolerant systems that have to move computations and data from one cell to another.

This EPS is written based on experience with a preliminary iWarp array compiler that was developed at Carnegie Mellon University. It tries to follow the EPS of the (single-cell) linker where possible and borrows some text and organization from the linker EPS.

1.4 Product identification

The iWarp combiner **iwcb** executes on a SUN host and produces code for an iWarp array. It is thus a part of the iWarp chain of cross development tools. The **-V** option of the combiner forces the combiner to emit a line with the current release number, thus disclosing the exact identity of the combiner in use. A preliminary version of the array combiner has been in use at Carnegie Mellon since August 1990.

1.5 Assumptions about iWarp systems

The iWarp combiner assumes it is working on a logical $m \times n$ grid or torus. The iWarp array loader later maps this logical array to a physical iWarp torus. The combiner provides command line arguments to specify the size of the logical array. The array combiner does not perform any physical array specific checks. Such checks are delayed to the iWarp array loader.

The execution model of the combiner is as follows: after the combiner has produced an iWarp array module, this module can be input to another combiner operation (e.g. if a module for a *2* x *3* array is combined with a module for a *2* x *5* array to form a module for a *2* x *8* array), or it can be input to the array loader **iwload** for activation on a suitable iWarp system. The iWarp array loader deals with all issues that arise from the the choice of the physical array for the execution of this array module. In the first step of execution on an iWarp system each cell performs initialization. In a traditional uniprocessor environment, this step can include an open for standard input and standard output, etc. In the multi-cell environment of an iWarp system, this step includes the initialization of logical connections between the processes executing on different cells. These connections are 'software connections' – they are realized by the runtime system and the communication agent of the iWarp cells. The information for this part of initialization (establishing connections) comes from the array combiner and is executed by the loader/runtime system.

In addition, the combiner assumes the existence of a runtime system. This runtime system provides services for connection (pathway) setup and management that are invoked by code generated by the combiner.

1.6 Assumptions about the combiner host system

The iWarp combiner will invoke various parts of the iWarp program development tool chain. Therefore, the complete iWarp tool chain must be available on the machine on which the combiner is executed. In addition, the iWarp combiner may invoke a C compiler to compile the array program, so it can be executed to create the array module. Therefore, the existence of (ANSI or K&R) C compiler on the combiner host is assumed.

1.7 Functions not described or provided

This EPS does not describe the iWarp array loader, which interacts closely with the iWarp combiner. The EPS of the iWarp array loader is forthcoming in a separate document.

1.8 Development phases

Releases of the iWarp combiner are synchronized with the development of iWarp systems.

1.9 Relation to other iWarp software

Figure 4 shows how the array combiner **iwcb** relates to other iWarp program development tools.

1.9.1 iWarp combiner and iWarp archiver

The iWarp combiner relies on the iWarp archiver to combine the cell programs and network descriptions into a single array module. Other tools extract information from the array module by using the iWarp archiver.

1.9.2 iWarp combiner and iWarp array loader

The iWarp combiner produces an array module for the iWarp array loader **iwload**. This file contains the single cell object files and network initialization information. The array loader creates initialization tables from the network initialization information, links the necessary system libraries with the cell programs, and activates the array program.

The array loader checks that the array module can be executed on the specified physical iWarp array. The array combiner contains no physical array specific checks. These checks are made by the array loader or later tools.

Any connections that have not been resolved at combine time must be resolved at load time. One example is a connection to an auxiliary cell. Since the location of auxiliary cells depends on the particular physical array, the connections to auxiliary cells must be resolved by the array loader.

1.9.3 iWarp combiner and iWarp module display

The **iwmd** command displays the information in an iWarp array module on an X window. By pointing at an array object, **iwmd** displays information about that object in the message window. By pointing at a cell, the files loaded on that cell are listed. By pointing at a port, the port name and associated network identifier are displayed. In addition, all connections between cells in the array module are displayed.

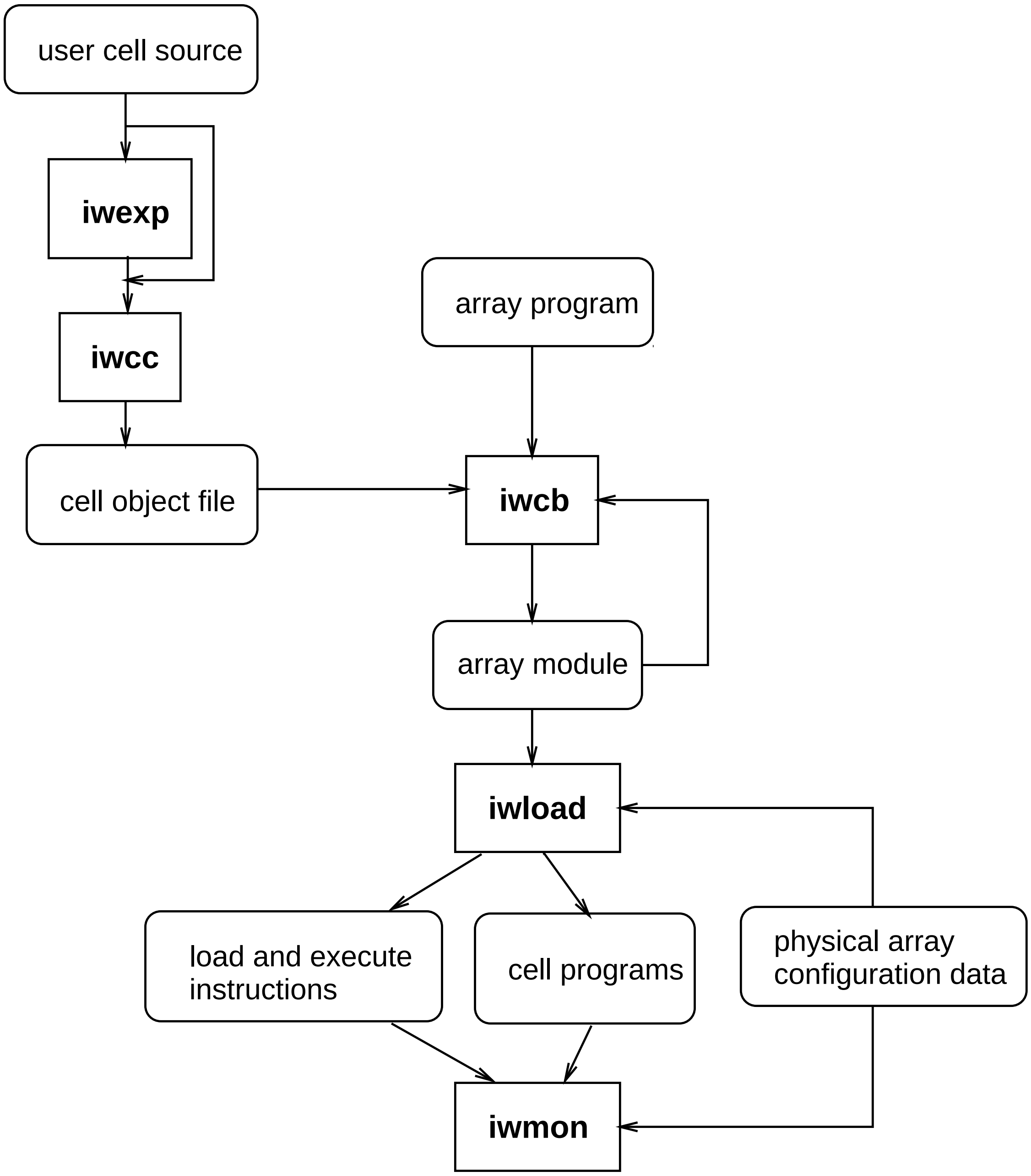

Figure 4: Flow chart of the iWarp tool chain

2 Purpose and general functions of the combiner

This section reviews the general functions of the iWarp combiner **iwcb** and outlines how a user communicates specific requests to the combiner. The user passes requests through command line arguments as well as through a iWarp array program. The command line arguments determine simple options (e.g. the number of cells in a regular torus), whereas the array program describes more involved requests like the connections between the different cells.

2.1 iWarp combiner functions

iwcb performs the following functions:

1. Combine one or more cell programs for individual cells into an array module.
2. Combine one or more cell programs for individual cells with one or more array modules to form a new array module.
3. Combine multiple array modules to form a new array module.
4. Determine how the program connections are routed between different cells in an array module. The routing is determined by
   - explicit routes provided by the user
   - the default routing, based on the hardware configuration and the allocation of logical channels.
5. Produce network descriptions tables that are used by the array loader to construct initialization tables for each cell of the array to set up the program connections as determined above. These tables also ensure that at set up time, the state of the cell programs is initialized correctly so that these connections can be identified and used by the cell program.
6. Provide a library of functions to describe the program connections of the iWarp application.

The input to the combiner is described in the Section 2.2; the output of the combiner is described in Section 2.3.

2.2 Inputs

The primary input to the combiner is the array program. This file defines

- the mapping of cell programs to cells in the target iWarp system
- the inclusion of previously defined array modules
- ports used by cell program to access networks
- program connections between the cells of the target iWarp system

Note that the combiner takes a *static* view of the iWarp system. A specific configuration is described as input, and the combiner tries to use this configuration as the base for the program connections. No attempt is made to change the configuration to better suit the current program connections. Here is an example to illustrate this limitation.

For every connection, a route is specified either explicitly or implicitly by the default routing scheme. For a collection of connections, it is quite possible that these connections (with their current routes) requires more communication resources in a cell than there are available. However, different routes for these connections may not have this problem. In such a case, the iWarp combiner will report an error; it will *not* attempt to find a better set of routes.

2.3 Outputs

The output of the combiner consists of an array module which contains

- a mapping for cell programs to iWarp cells
- descriptions of program network connections
- the included cell program object code

The cell program mapping and the program connection descriptions are used by the iWarp array loader. From this the array loader creates initialization tables which are interpreted by initialization code executed on the iWarp cells before the execution of the application begins.

3 Combiner command line options

3.1 Sample iwcb command

The command shown below assumes that there is an array program `foo.desc.c`, in the format introduced in Section 4. The combiner is asked to leave the output in a file with the name `foo`. This file will be generated only if the combine step encounters no errors.

```
iwcb -o foo foo.desc.c
```

The combiner invokes the C compiler on the development platform to compile array program and link it with the combiner library **libcb.a**. The resulting executable creates tables that are interpreted by the array loader to produce the array load and initialization maps.

3.2 iwcb options

-V Print the version number and continue.

-VV Print the version number, prints usage information, and exit.

-d Enable the printing of additional debugging statements.

-row *#* Set the number of rows in the grid. Defaults to 8.

-col *#* Set the number of columns in the grid. Defaults to 8.

-w Allow routes to include the backedges of the torus. By default only the grid edges are used.

-M*dir* Add *dir* to the list of directories searched while looking for included array modules. By default **iwcb** searches the current directory and **/iwarp/lib/modules**.

-I*dir* Add *dir* to the list of directories searched while looking for include files. By default **iwcb** searches **/iwarp/include**.

-L*dir* Add *dir* to the list of directories searched while looking for libraries. By default **iwcb** searches **/iwarp/lib**.

-o *output* Set the name of the array module to *output*. Defaults to *array.out*.

array program User C program that describe the array's communication networks and cell program to cell assignments.

3.3 iwcb environment variables

By default the combiner library **libcb.a** is located in **/iwarp/lib**. If you wish to use another copy of **libcb.a** set the environment variable **CBLIB** to the new location. Otherwise, you can use the **-L** option to have **iwcb** look for **libcb.a** in a different directory. If you want **iwcb** to look in non-standard include directories while it is compiling, use the **-I** option or set the environment variable **CBINCDIR** to the new include directories prefixed by the **-I** flag. To use a non-default version of the runtime system, set the environment variable RTS_VERSION to the desired version number.

3.4 Sample iwmd command

The command below will display the logical array and networks defined in `foo`.

```
iwmd foo
```

By pointing at ports and cells, more information about the programs loaded and the ports defined is displayed at the top of the display.

3.5 iwmd options

iwmd only takes one argument, which is the name of an array module.

4 Specifications of array modules

The combiner needs three kinds of information to set up the communication network for an application: the allocation of logical channels to physical busses (how many logical channels are multiplexed over each physical bus), the names of the object modules to be executed on the cells, and the program connections between the cells. Each program connection requires a route, which establishes the physical path used to transport the data through the iWarp system, and the names of the local ports, which are used by the program executing on the source and destination cell to access the connection. Furthermore, the iWarp communication system allows a program to link different connections ('pathways') to form a network. Networks reserve communication resources for a class of messages and allow one cell to send data to a number of cells without requiring an explicit connection to each of them. All this information is described in the array program for the combiner, separate from the individual cell programs.

There exist two options to describe the connections: either design a separate 'component description language' (and then require the user to specify the connections in this language) or use a library with an existing programming language. Both approaches have advantages and disadvantages. In the interest of limiting the number of languages a user has to master to program an iWarp system, the iWarp software adopted a solution that is based on the later approach.

The user (or program generator) provides a body of text that contains calls to system defined functions introduced later in this section that define connections and cell program mapping. The array program is coded in any suitable language (for which there exists a compiler on the program development platform that supports calls to the system defined utilities). In practice, the array program is coded in C.

The combiner compiles this array program and executes it. The functions invoked by the array program produce a set of tables and other information that is then processed by the combiner to decide how to set up the connections for this application. Note that the combiner is executed after linking but before process activation.

In the rest of this section, we briefly discuss cell naming and the various networks that can be realized by the iWarp communication system. Then we introduce the functions that allow the user to describe the connections between cell programs.

4.1 Cell naming conventions

The iWarp array tool chain works with two types of cells: array cells and auxiliary cells. The array cells represent the regular iWarp array cells, and they make up a logical array, either a torus or a grid. The logical array cells are addressed using the cartesian coordinates as shown in Figure 5. The cartesian coordinates are used by the programmer (or program generator) in the cell program and the array program.

The auxiliary cells are additional special cells like the host interface cells and disk interface cells. These cells are not part of the regular physical array. They are attached to the edge of the physical array, and their actual location is system dependent. The array program and cell programs address the auxiliary cells as if they are part of a logical list. See the last row in Figure 5 for an example of addressing the logical list of auxiliary cells.

The actual addresses used by the iWarp array are not relevant to the application program as long as there is a mapping from cell names to addresses that is managed by the combiner in collaboration with the runtime system.

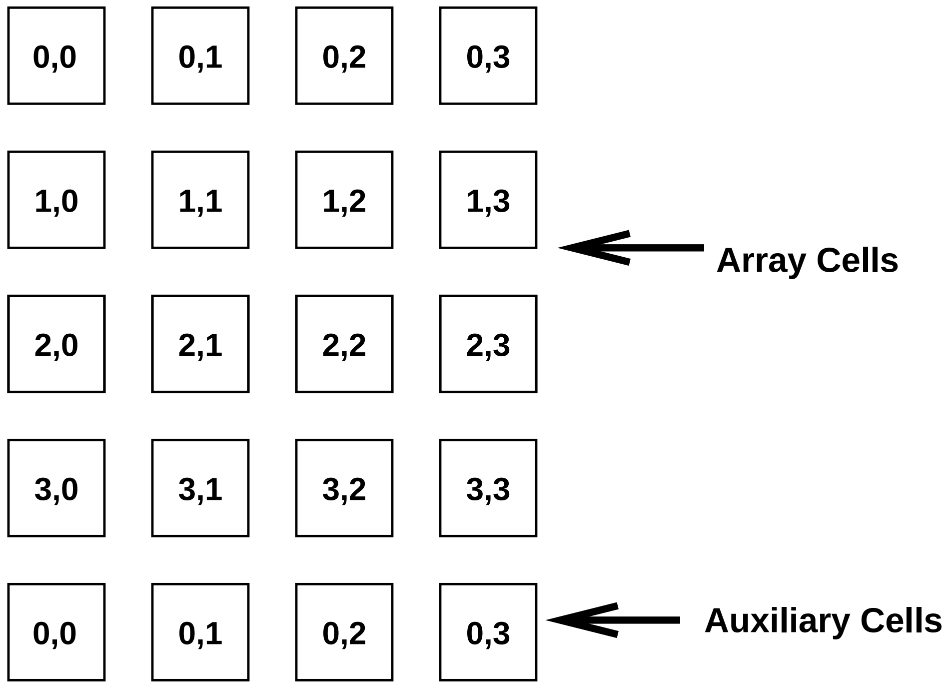

Figure 5: Example of naming in a 4 $\times$ 4 logical array with 4 auxiliary cells.

4.2 iWarp networks

An iWarp *network* consists of cells and pathways between these cells. Networks are directed, and they can include cycles. Figure 6 shows an example of a simple network; cells that are part of the network are shaded gray and black lines indicate the pathway segments. Through this network, cell (0,0) can transmit data to cells (0,1) and (1,3), and cell (0,1) can send data to cell (1,3), but no other transmissions are possible.

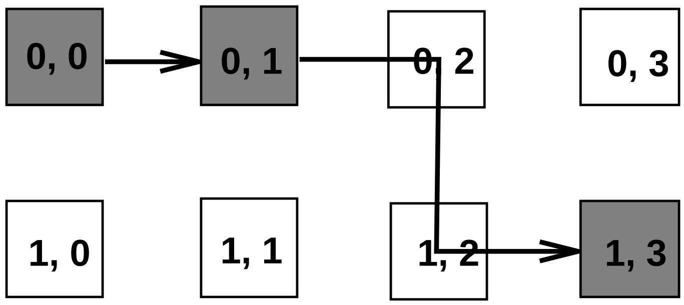

Figure 6: Simple iWarp network

A cell can be part of multiple networks, subject to resource constraints in the communication agent. The transmission of data through a network is accomplished by the communication agents of the cells that built up a pathway. The source cell injects the data, and the destination cell retrieves the data, but no further program intervention is required. That is, programming overheads are associated only with the source and destination cells. Computation in the other cells can proceed without interference or intervention.

The information about the shape of a network is only stored locally. That is, each cell needs to know about the pathway segments that end at this cell as well as which pathway segments start at this cell. No global controller is required to maintain the network. Of course, a cell that wants to transmit data to another cell via a network has to know this cell's address, but no global network information needs to be maintained.

4.2.1 Network restrictions

Figure 7: Example an illegal network

If you consider the cells in an iWarp array as cartesian grid points, you can think of a network as a directed graph on the cartesian points. The array combiner and related tools only support a subset of the possible networks. The combiner only supports networks that are paths on the cartesian grid, because the routing of data on such networks can be taken care of entirely by the communicatino agent. Consider the illegal network in Figure 7. A message arriving at cell (0,2) must decide whether to continue right or go down. Such a decision cannot be made without software intervention.

The programmer can simulate networks of higher degree in software. At each decision point, like at cell (0,2) in Figure 7, a software function could read some routing information and direct the message in the correct direction. Dynamic connections are a more efficient way to support networks of higher degree. By reserving resources in the proper direction on each cell, the program can create a dynamic connection for each message, and the hardware will take care of routing the message to the proper destination. If the array program reserves 1 logical channel in each direction on each cell, dynamic connections can be created along the total torus network in Figure 8

The most elementary network consists of a single pathway. That is, only two cells participate, the source cell and the destination cell. The pathway might consist of multiple segments that involve other cells, but these cells cannot use the network. Such a network establishes a point to point connection. Figure 9 shows an example of a point to point network.

Intermediate cells that the network passes through can tap into the network. These intermediate cells can read from or write to the network. Data not addressed to the intermediate cell will express through the cell without interrupting the computation agent. In Figure 6 cell (0,1) has a tap on the network.

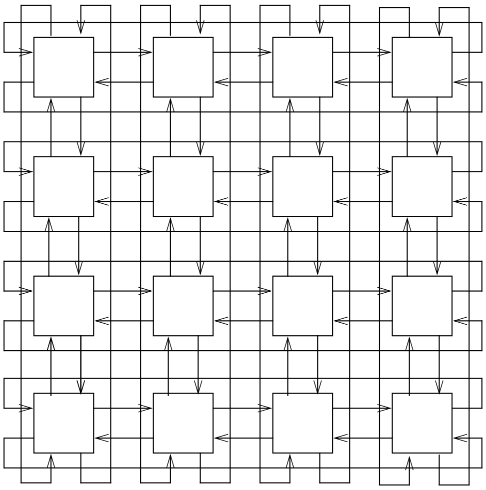

Figure 8: Network that covers all cells of the iWarp torus

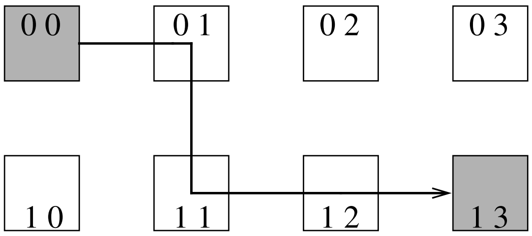

Figure 9: Pathway connecting iWarp nodes

The iWarp communication agent provides the application programmer or system builder with numerous options to form networks. To guarantee that all messages sent from one cell to another cell arrive at the destination in the order in which the message were generated, we impose the the constraint that at each cell no two pathways of the same network can multiplex over the same bus. That is, for each network, two adjacent cells are connected by at most *one* pathway segment. Figure 10 shows a network that does not meet this constraint, since two pathway segments (black arrows) are multiplexed over the same physical bus between the second and third cell.

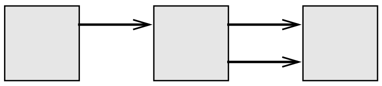

Figure 10: Illegal iWarp network

The above restriction minimizes the amount of network information that must be carried along between two adjacent cells. In practice, it does not limit the topology choices unduly. Figure 11 depicts legal usages. Note that there are separate input and output busses and therefore no two pathway segments are multiplexed over the same bus in the network shown in the lower part of Figure 11. This network enables cells (0,0), (0,1) and (1,1) to transmit data to all other cells that are part of the network.

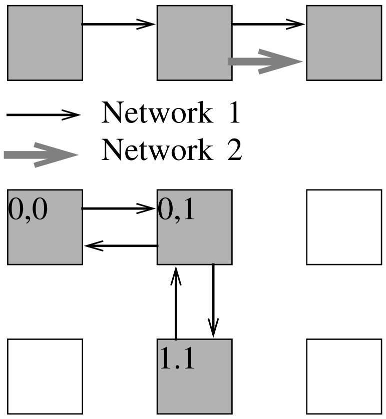

Figure 11: Legal iWarp networks

4.3 Array module hierarchy

In addition to being executed directly, array modules can be included in new array modules. There are two ways you can think about combining array modules: *overlaying* and *nesting*. The difference between these models deals with the cell addressing that cell programs use within a particular module. In the overlay model, when an array module is inserted, the cell programs in included array module use the cell naming of the external module. Consider placing array module B in Figure 12. Array module B is placed at cell (2,2) in a 4×4 array. In the overlay model, the cell program on cell (0,0) in array module B now considers itself on cell (2,2). This model is useful for loading generic programs like a package for synchronization or for loading predefined networks.

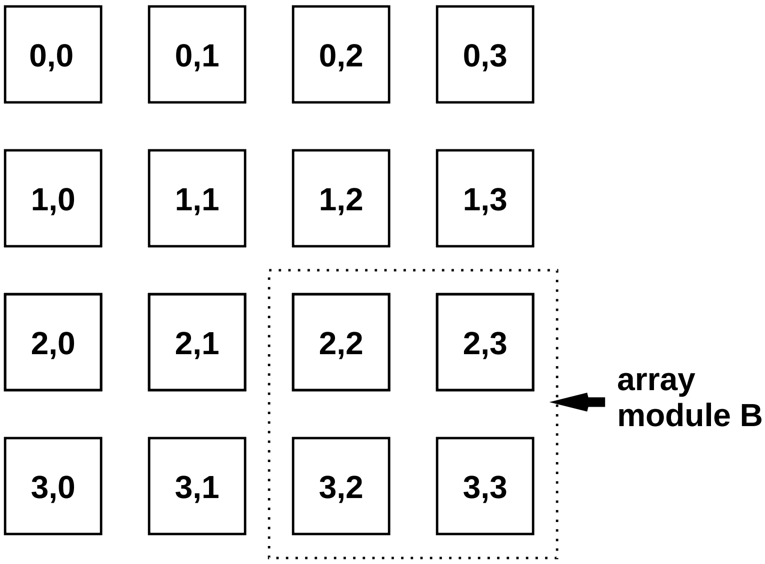

Figure 12: Array module B placed on a 4 by 4 array.

The nesting interpretation is necessary to use array modules as building blocks. In the nesting interpretation, the cells in the included array module maintain their original cell identity. In the example in figure 12 the cell program on (0,0) in array module B still considers itself on cell (0,0). If that cell program sends to the cell to its right, it still sends to cell (0,1). In this model the cell program on a cell uses the logical cell ID of the most deeply nested array module that encloses the cell.

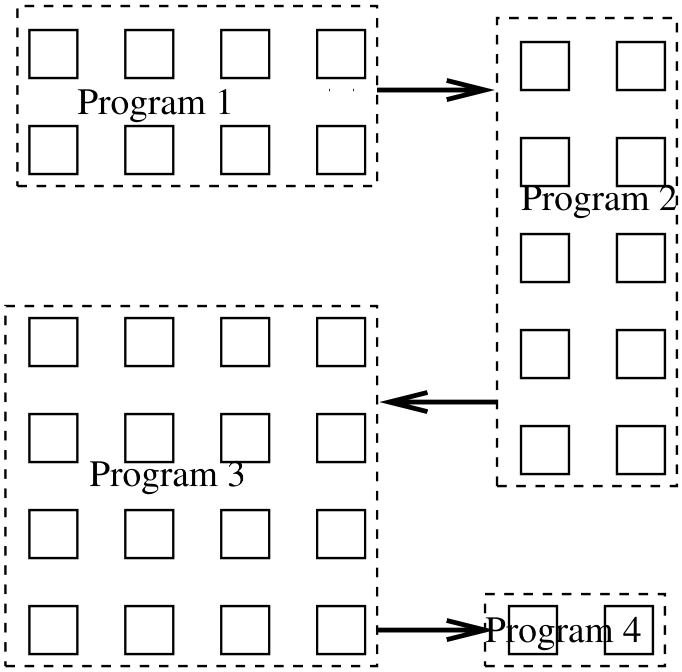

Figure 13: Four array modules pipelined together by including instances of the array modules in a 6×6 array module and connecting external ports.

Some sense of an "external" port is also needed to use array modules as building blocks. When writing to or reading from an external port, the cell program is communicating with a program outside its locally defined array module. By connecting external ports, one can create a pipeline of different programs. For instance, one could pipeline a matrix operation program, a signal processing program, and an image processing program. See figure 13 for a picture of four separately defined array modules pipelined together.

This nesting interpretation is much like the hierarchical model used in computer aided design. For instance while defining an ALU, you may first define an adder. To do this you would create a module that defined the adder and specify set of inputs and outputs for the module. Later in your design, you could use the adder as a black box by properly using the inputs and outputs. Similarly in our model, you can define an array module that performs matrix multiplication and specify some input and output external ports. Later you can use the matrix multiply module in another program as a black box by properly using the input and output external ports.

In order to cleanly integrate the idea of array module hierarchies into our network definition functions, consider the following model for defining array modules. All array modules are defined from already existing modules. Each cell is an instance of the lowest level module. The instance ID of a cell is predefined to be the row and column cell ID you would expect, so a cell ID is a special case of an instance ID. When defining a new array module, you can create new instances of already defined array modules in the new array module. Each created array module instance is referred to by an instance ID. From the instance ID you can access the external ports defined in that array module.

4.4 Specifications in C

Array programs call the following functions to define program networks and program placement. All array programs should include the file **cb_def.h**. The main function in the array program must be called `user_main`. Two integer arguments are passed to `user_main` from **iwcb**, the row size and column size of the logical array.

4.4.1 Logical channel reservation

Logical channels are the hardware resources used to construct networks. To ensure that networks get set up as expected a sufficient number of logical channels must be reserved on each cell. The combiner takes care of this automatically for static networks defined in the array program. For message passing and other dynamic networks, the programmer must reserve the appropriate resources directly.

```
int logical_channel_reserve(cell, num_left, num_right, num_up,
                            num_down, num_out, pool)
cellid_type *cell;
int num_left;
int num_right;
int num_up;
int num_down;
int pool;
```

This function reserves logical channels on `cell`. It reserves `num_left`, `num_right`, `num_up`, and `num_down` logical channels to the left, right, up and down respectively. It also reserves `num_out` logical channels for output. It reserves these channels in pool `pool`. The combiner allocates logical channels for its static buffers in pool 0. The user specifies either PCS_POOL0 or PCS_POOL1. Both pools can be used to implement deadlock free routing algorithms.[1]

4.4.2 Program connections

There are two parts to creating a network. First ports are created on cells. Then a network is created between the ports. Additional ports can be connected to the network on intermediate cells so those cells can tap into the network.

```
port_type *input_port_create(port_name, cell)
char *port_name;
cellid_type *cell;

port_type *output_port_create(port_name, cell)
char *port_name;
cellid_type *cell;
```

[1] Currently can only reserve buffers in pool 0. Ability to reserve in both pools will come in the next version of RTS.

These functions create a port of the specified type. A pointer to the new port is returned. The returned port is not connected to any network or mapped to any external port. At runtime, the cell corresponding to the cell pointer can retrieve this port by its name `port_name`.

```
network_type *network_create(src_port, dst_port, route)
port_type *src_port;
port_type *dst_port;
route_type *route;
```

`network_create` creates a network from port `src_port` to port `dst_port`. A pointer to the new network is returned. The new network has `src_port` and `dst_port` connected to either end. A connected port can access the network at runtime. `route` is a linked list of type `route_type`. It indicates the route the network should follow. If `route` is NULL, a default route is used. The default route goes along the row first and then along the column.

The route list must include the source cell, destination cell, and all intermediate cells where the route turns. For example, to specify a route from cell (0,0) to (1,2) turning at cell (1,0), the linked list would contain cells (0,0), (1,0), and (1,2). Below are several functions that build up routes.

```
route_type* make_route(cell, route)
cellid_type *cell;
route_type *route;

route_type *make_directed_route(cell, direction, route)
cellid_type *cell;
int direction;
route_type *route;

route_type *add_to_route(cell, route)
cellid_type *cell;
route_type *route;

add_to_directed_route(cell, direction, route)
cellid_type *cell;
route_type *route;
```

The route construction functions come in two flavors. The `make_route` functions build up routes by adding new cell IDs to the beginning of the list. The `add_to_route` functions build up routes by adding new cell IDs to the end of the list. `make_route` and `add_to_route` calculate the direction the route should go. For grids there is no choice about the direction along a row or column to the next entry. For a torus, there is a decision about whether to use the backedge. In the torus case `make_route` and `add_to_route` pick the direction that results in the least distance between the cells. `make_directed_route` and `add_to_directed_route` let the user specify the next direction of the route. Legal directions are `LEFT`, `RIGHT`, `UP`, `DOWN`, and `SELF`.

Here are some examples of specifying a route from (0,0) to (7,7) on a 8×8 grid.

```
route = make_route(cell(0,0), make_route(cell(7,0),
```

```
                                        make_route(cell(7,7), NULL)));
```

```
    route = add_to_route(cell(0,0), NULL);
    route = add_to_route(cell(7,0), route);
    route = add_to_route(cell(7,7), route);
```

Suppose the route is specified on a 8 by 8 torus. To route across the face of the array, the direction must be specified.

```
    route = add_to_directed_route(cell(0,0), DOWN, NULL);
    route = add_to_directed_route(cell(7,0), RIGHT, NULL);
    route = add_to_directed_route(cell(7,7), SELF, NULL);
```

When either the source or destination cell is an auxiliary cell, the user program cannot define a route. A default route must be used, so the `route` parameter must be NULL, because the location of the auxiliary cell depends on the particular physical array.

```
network_type *network_join(nw1, nw2)
network_type *nw1;
network_type *nw2;
```

`network_join` creates a new network out of two already existing networks. The resources used by `nw1` and `nw2` are absorbed by the new network, so `nw1` and `nw2` are not valid networks after this call.

```
int port_connect(port, nw)
port_type *port;
network_type *nw;
```

`port_connect` connects the previously defined `port` to the previously created network `nw`. In order to connect `port` the network must pass through the cell `port` is created on. Once a port is connected to a network, that port can access the network at runtime.

```
int external_port_map(port, external_port_name)
port_type *port;
char *external_port_name;
```

This function maps `port` to an external port in the current array module called `external_port_name`. `port` can be either a local or external port. `port` should not be connected to a network or already mapped to an external port in the current array module. If the current array module is included in another array module `port` can be referenced by the name `external_port_name`. Only external ports defined in the outermost array module can be referenced when that array module is included in another array module.

```
port_type *external_port(external_port_name, instance)
char *external_port_name;
instance_type *instance;
```

This function returns a pointer to the external port named `external_port_name` defined in the array module referenced by `instance`. The external port pointer can be used as an argument to `network_create`, `port_connect`, and `external_port_map`.

4.4.3 Object placement

These functions map cell programs and previously defined array modules to cells.

```
int place(instance, prog_name)
instance_type *cell;
char *prog_name;
```

This function maps the cell program named `prog_name` to `instance`. If instance is not a cell ID, it maps the program to the logical (0,0) cell in the instance array module. The code corresponding to `prog_name` is stored in the resulting array module.

```
int overlay(offset, module_name)
cellid_type *offset;
char *module_name;
```

This function includes the previously defined array module named `module_name` at `offset` using the overlay model described in section 4.3. This function is useful to include predefined networks.

```
instance_type *instance_create(offset, instance_name, module_name)
cellid_type *offset;
char *instance_name;
char *module_name;
```

This function creates an instance of the array module named `module_name` at `offset`. The array module is included following the nesting interpretation described in section 4.3. An instance pointer is returned that refers to the new array module instance. In the cell program this instance pointer can be retrieved by using the name `instance_name`.

Both `overlay` and `instance_create` look for the requested array module in several directories. First they look in the directories specified by **-M** arguments to the command line. Then they look in the current directory and then in `/iwarp/lib/modules`.

5 Related documents

- *iWarp linker EPS*, Herbert G. Mayer, Version 0.5 of March 16, 1990. Intel internal document.

  Describes the general behavior of the iWarp single-cell linker **iwld**.

- *iWarp runtime system EPS* ??

- *iWarp programmed communication service (PCS) data movement EPS*, Thomas Gross et al., Version 1.0 of May 28, 1990. Carnegie Mellon/Intel internal document.

  Describes the functions used by C programs for *user programmed* inter-processor communication. Build on top of the raw PCS.

- *iWarp programmed communication service (PCS) combiner design document*, TBD, Version X.X of YYYY ZZ, 1990. Carnegie Mellon draft document.

  Describes the internal workings of the iWarp combiner.

- *iWarp array loader EPS*, TBD, Version X.X of YYYY ZZ, 1990. Intel internal document.

- *iWarp array loader design document*, TBD, Version X.X of YYYY ZZ, 1990. Intel internal document.

A Program examples

A.1 Three cell example

This is a simple send and receive example. Below are the listings of the user cell program, a network description, and a makefile. This program call be found in **/iwarp/example/three_cell**.

This is the user program for all three cells. It is in file **cellprog.c**.

```
#include <stdio.h>
#include <pcs_def.h>

#define DATA_SIZE 16

port_type *Xout, *Xin;

port_list_type ports[] = {
{ "Xout", &Xout },
{ "Xin", &Xin }};

main()
{
    int i;
    cellid_type *self, dst;
    int data[DATA_SIZE];

    pcs_init(ports, NUM_PORTS(ports), NULL, 0);

    self = get_cellid();
    if (self->col == 0) /* first cell */
    {
        for (i = 0; i < DATA_SIZE; i++)
    data[i] = i;

        systolic_gate_bind(Xout, GATE0);
        dst.type = ARRAY;
        dst.row = 0;     dst.col = 1;
        send_msg(Xout, GATE0, (char *)data, DATA_SIZE*sizeof(int), &dst);

        dst.col = 2;
        send_msg(Xout, GATE0, (char *)data, DATA_SIZE*sizeof(int), &dst);
        printf("Send messages\n");
    }
    else if (self->col == 1) /* middle cell */
    {
        systolic_gate_bind(Xin, GATE0);
        receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
        printf("Received message\n");
```

```
        systolic_gate_bind(Xout, GATE0);
        dst.type = ARRAY;
        dst.row = 0;       dst.col = 2;
        send_msg(Xout, GATE0, (char *)data, DATA_SIZE*sizeof(int), &dst);
        printf("Send message\n");
    }
    else /* last cell */
    {
        systolic_gate_bind(Xin, GATE0);
        receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
        receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
        printf("Received messages\n");
    }
    pcs_de_init();
    exit(0);
}
```

The array program defines the communication networks for the array module and assigns user code to cells. This is in file **arrayprog.c**.

```
#include <cb_def.h>
#include <stdio.h>

user_main()
{
    network_type *nw;
    cellid_type src, dst;
    port_type *src_port, *dst_port;

    src.type = dst.type = ARRAY;
    src.row = src.col = dst.row = 0;
    dst.col = 2;

    src_port = output_port_create("Xout", &src);
    dst_port = input_port_create("Xin", &dst);

    nw = network_create(src_port, dst_port, NULL);

    place(&src, "cellprog.o");
    place(&dst, "cellprog.o");

    src.col = 1;
    src_port = output_port_create("Xout", &src);
    dst_port = input_port_create("Xin", &src);
    port_connect(nw, src_port);
    port_connect(nw, dst_port);
    place(&src, "cellprog.o");
```

```
}
```

This makefile ensures that the cell programs have been preprocessed and compiled before **iwcb** is called. This preprocessing step can also be skipped. **iwload** will load and execute the array module on the iWarp array attached to sib 1 on host victory.

```
.SUFFIXES: .i
TARGET_ARCH=iwarp
CC= iwcc
INC= -I/usr/iwarp/pcs2.0/include
CFLAGS= -DIWARP -DIWARP_HW -DIWARP_CELL -DRTS -v$(IWARP_ARCHITECTURE) $(INC)

all:    cellprog.o two_cell
        iwload tree_cell -t victory 1

three_cell:     arrayprog.c
        iwcb -row 1 -col 3 -o tree_cell arrayprog.c

# Uncomment these two targets to run preprocessing
#cellprog.i:    cellprog.c
#       iwpp cellprog.c > cellprog.i

#cellprog.o:  cellprog.i
#       $(CC) $(CFLAGS) -c cellprog.i

clean:
        /bin/rm -f *.o *.i two_cell-* two_cell_* two_cell std[oe]* go* cb1
```

A.2 SIB example

This is a simple send/receive program that involves the SIB cell. The SIB (the first auxiliary cell in this case) sends data to cell (0,0), and cell (0,0) passes this data to cell (0,1). Below is the cell program for all three cells. This example can be found in **/iwarp/example/sib**.

```
#include <pcs_def.h>
#include <stdio.h>

#define DATA_SIZE 16

port_type *Xout, *Xin;

port_list_type ports[] = {
{ "Xout", &Xout },
{ "Xin", &Xin }};

main()
```

```
{
    int i;
    cellid_type *self;
    int data[DATA_SIZE];

    pcs_init(ports, NUM_PORTS(ports), NULL, 0);

    self = get_cellid();

    if (auxiliary(*self)) /* SIB */
    {
        for (i = 0; i < DATA_SIZE; i++)
            data[i] = i;
        systolic_gate_bind(Xout, GATE0);

        /* send to cell (0,0) */
        send_msg(Xout, GATE0, (char *)data, DATA_SIZE*sizeof(int), NULL);
        printf("Send message\n");
    }
    else if (self->col == 0) /* sending cell */
    {
        systolic_gate_bind(Xin, GATE0);
        /* receive data from SIB */
        receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));

        systolic_gate_bind(Xout, GATE0);
        send_msg(Xout, GATE0, (char *)data, DATA_SIZE*sizeof(int), NULL);
        printf("Send message\n");
    }
    else
    {
        systolic_gate_bind(Xin, GATE0);
        receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
        printf("Received message\n");
    }
    pcs_de_init();
    exit(0);
}
```

Below is the array program file for this example.

```
#include <cb_def.h>
#include <stdio.h>

user_main()
{
    network_type *nw;
```

```
    cellid_type src, dst;
    port_type *src_port, *dst_port;

    src.type = dst.type = ARRAY;
    src.row = src.col = dst.row = 0;
    dst.col = 1;

    src_port = output_port_create("Xout", &src);
    dst_port = input_port_create("Xin", &dst);

    nw = network_create(src_port, dst_port, NULL);

    place(&src, "cellprog.o");
    place(&dst, "cellprog.o");

    dst = src;
    src.type = AUX;
    src.col = 0;   /* first auxiliary cell */

    src_port = output_port_create("Xout", &src);
    dst_port = input_port_create("Xin", &dst);

    nw = network_create(src_port, dst_port, NULL);

    place(&src, "cellprog.o");
}
```

The makefile from the previous example can also be used here.

A.3 Spooling example

This is a spooling version of the first two cell program. Notice that the open and close message markers are still sent systolically. Only the cell program is different. The same makefile that was used in previous examples can be used here. This example can be found in **/iwarp/example/spool**

```
#include <stdio.h>
#include <pcs_def.h>

#define DATA_SIZE 16

port_type *Xout, *Xin;

port_list_type ports[] = {
{ "Xout", &Xout },
{ "Xin", &Xin }};

main()
```

```
{
    int i;
    cellid_type *self;
    int data[DATA_SIZE];

    pcs_init(ports, NUM_PORTS(ports), NULL, 0);

    self = get_cellid();
    if (self->col == 0) /* sending cell */
    {
        for (i = 0; i < DATA_SIZE; i++)
            data[i] = i;

        systolic_gate_bind(Xout, GATE0);
        /* systolically send open message */
        send_open_msg(Xout, GATE0, NULL);

        /* start spool */
        send_block_bg(Xout, PCS_SPOOL0, (char *)data, DATA_SIZE*sizeof(int));

        /* wait for spool to finish */
        await_bg_complete(Xout, PCS_SPOOL0);

        /* systolically send close message */
        send_close_msg(Xout, GATE0);
        printf("Send message\n");
    }
    else
    {
        systolic_gate_bind(Xin, GATE0);
        /* systolically receive open message */
        receive_open_msg(Xin, GATE0);

        /* Start spool */
        receive_block_bg(Xin, PCS_SPOOL0, (char *)data, DATA_SIZE*sizeof(int));

        /* wait for spool to finish */
        await_bg_complete(Xin, PCS_SPOOL0);

        /* systolically receive close message */
        receive_close_msg(Xin, GATE0);
        printf("Received message\n");
    }
    pcs_de_init();
    exit(0);
}
```

The array program defines the communication networks for the array module and assigns user code to cells. This is in file **arrayprog.c**.

```
#include <cb_def.h>
#include <stdio.h>

user_main()
{
    network_type *nw;
    cellid_type src, dst;
    port_type *src_port, *dst_port;

    src.type = dst.type = ARRAY;
    src.row = src.col = dst.row = 0;
    dst.col = 1;

    src_port = output_port_create("Xout", &src);
    dst_port = input_port_create("Xin", &dst);

    nw = network_create(src_port, dst_port, NULL);

    place(&src, "cellprog.o");
    place(&dst, "cellprog.o");
}
```

A.4 More examples

Additional iWarp example programs can be found in the directory **/iwarp/example** on the iWarp Sun network.