INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-415-642-4274 • FAX 1-415-643-7684

Ring Array Processor (RAP): Software Architecture

Jeff Bilmes Phil Kohn

TR-90-050

January 1991

Abstract

The design and implementation of software for the Ring Array Processor (RAP), a high performance parallel computer, involved development for three hardware platforms: Sun SPARC workstations, Heurikon MC68020 boards running the VxWorks real-time operating system, and Texas Instruments TMS320C30 DSPs. The RAP now runs in Sun workstations under UNIX and in a VME based system using VxWorks. A flexible set of tools has been provided both to the RAP user and programmer. Primary emphasis has been placed on improving the efficiency of layered artificial neural network algorithms. This was done by providing a library of assembly language routines, some of which use node-custom compilation. An object-oriented RAP interface in C++ is provided that allows programmers to incorporate the RAP as a computational server into their own UNIX applications. For those not wishing to program in C++, a command interpreter has been built that provides interactive and shell-script style RAP manipulation.

1 Introduction

This document describes the Ring Array Processor (RAP) software architecture design, implementation, and system evolution. A user's manual for the RAP is available [KB90], along with reports on the system architecture [Mor90] and hardware implementation [Bec90].

The Ring Array Processor [MBK+90] [Mor90] is a multi-DSP system targeted for speech recognition using connectionist algorithms. A RAP system consists of 1 to 16 9U VME bus boards, each of which contains 4 Texas Instruments TMS320C30 DSP chips running at 16MHz [Tex88a]. Each RAP processing node consists of a TMS320C30 and 4MB or 16MB of dynamic RAM (with either 1 or 4 Mb chips respectively), 1MB of fast static RAM, and 8KB of very fast on-chip RAM. The theoretical peak performance is 128 MFLOPS per board, and test runs of algorithms of interest show a sustained throughput roughly one-third to one-half of this.

The RAP was designed for a particular subset of layered artificial neural network algorithms. The initial use of the RAP is for the back-propagation algorithm [RHW86] used as part of a probability estimator for a Hidden Markov Model speech recognizer [MB90]. When this algorithm is partitioned in a particular way, a SIMD style of programming parallel machines is quite suitable. Additionally, for these algorithms shared memory between nodes is not necessary. The RAP has no shared memory between nodes but still supports a MIMD style of programming. All communication between the TMS320C30 processors (or nodes) of a RAP board and between boards in a RAP system is done through a unidirectional ring that is independent of the VME bus where the RAP resides. The RAP is similar to the WARP [PGTK88] and other systolic array processing machines but differs in the nature of the communication process.

The three primary instructions used to communicate between nodes are `ring_get`, `ring_put`, and `ring_shift`. `ring_get` reads a data word from the previous node, or blocks until one is ready. `ring_put` writes a data word to the subsequent node, or blocks until that node reads. `ring_shift` reads a data word from the previous node and writes that word to the subsequent node, or blocks until the previous node has written and the subsequent node has read. These routines provide all the RAP's synchronization primitives. The nodes may become unsynchronized during periods of no ring usage, but will re-synchronize upon initiation of ring usage.

The RAP ring and the VME memory interfaces were implemented using Xilinx Programmable Gate Arrays [Bec90]. While the hardware is fixed, these parts provide flexible ring semantics in that the ring parts may be re-programmed to provide different operations. For a 4 board system, the peak communication between nodes is 1024MB/sec, peak computation is 512 MFLOPS, the forward propagation step of the back-propagation algorithm attains 408 MFLOPS (256 units/layer), and the complete back-propagation algorithm attains 239 MFLOPS (256 units/layer, 1 hidden layer).

2 RAP Software Architecture Design Goals

Given the board described in the introduction, it was necessary to derive a set of goals to follow during the implementation process.

1. Computational Efficiency: get as close to the theoretical MFLOPS rating as possible. The primary purpose of the RAP is to be a computation server. Therefore, we didn't want the software to reduce performance in any way.

    - Write many inner TMS routines in assembly.
    - Minimize the interaction between the RAP and the controlling host during computation.

- Provide flexibility for where code and data may be placed. Let the user place code in TMS320C30 on-chip RAM even though it is not addressable by the host machine through the VME bus.

2. Easy to Use: given an existing RAP application, it should be possible for a naive user to quickly learn how to run it.

   - The RAP should have a straightforward user interface.
   - Simulate a UNIX[1] operating system shell closely but without providing gratuitous features.

3. Easy to Learn: given an existing computationally expensive UNIX application, it should be easy for a programmer to port it to the RAP.

   - Provide a familiar programming environment.
   - Give a wealth of library routines for doing low-level ring communication so the programmer will not need to learn the innards of the ring.

4. Flexibility: provide flexibility to programmers writing new RAP applications. We did not wish to write a full operating system for the RAP, but wanted to provide an easy way for a programmer to use RAP applications inside general UNIX programs (e.g. interactive graphics programs like a connectionist simulator, a UNIX filter using the RAP as a computational server, or anything else computationally expensive).

   - Provide an easy way for RAP programs to communicate with new and existing UNIX programs.
   - Provide features that will facilitate and encourage the writing of new RAP applications. Try to do this without writing a complex operating system.

5. ASAP: get things working in a reasonable amount of time. Typical runs of the back-propagation algorithm were taking two or three days to complete on a Sun Workstation. Therefore, we wanted to apply the RAP to real problems soon. We did not want to spend much more than six months of software development time.

   - Use the C compiler for the TMS320C30 [Tex89] provided by Texas Instruments for as much of the system related code as possible without sacrificing computational efficiency. Use TMS320C30 assembly coding only where necessary.
   - Delay the implementation of ideas for more complex node communication (such as general message passing, RPC, RAP multi-programming, and mail boxes). Only build what is necessary to get a running system satisfying the above goals.

3 The Three Hardware Platforms

Writing software for the RAP involved developing code for three hardware platforms: Sun Workstations, VxWorks/Heurikon MC68020 (described later), and the TMS320C30 (see figure 1).

Any UNIX application that manipulates the RAP is called a `rapClient`. A `rapClient` connects to a `rapServer` (or RAP host) which resides on the machine hosting a RAP system (e.g. if a RAP board is in a Sun VME bus backplane, a `rapServer` daemon process will wait for connections from

[1] UNIX is a trademark of AT&T Bell Laboratories

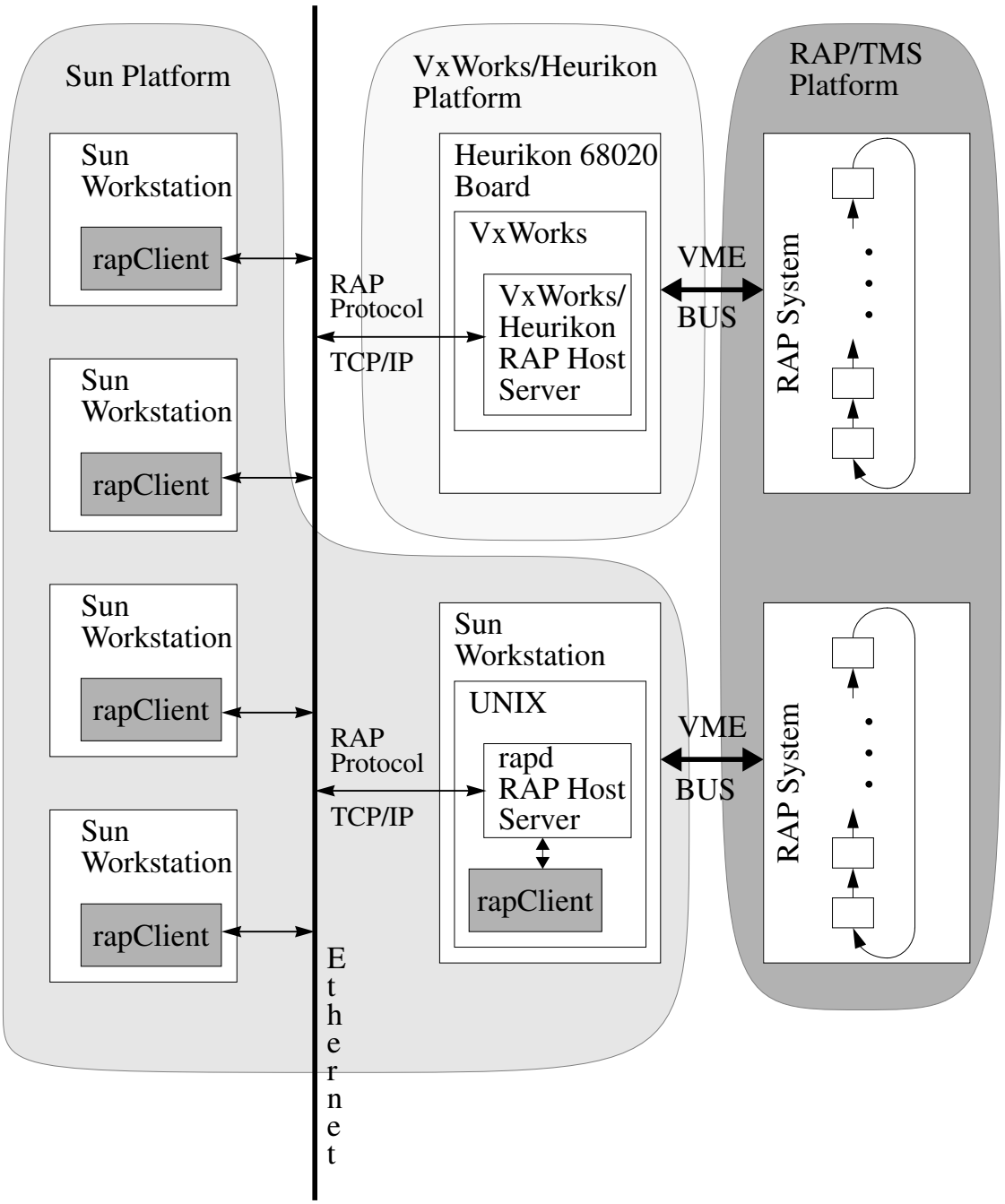


Figure 1: RAP software environment

rapClients and acknowledge interrupts from the RAP board). rapServer software runs both on VxWorks (written in C) and on UNIX workstations (written in C++ [ES90], and C for the device driver). A user may write RAP code and run it on either type of rapServer without recompiling. A rapServer may directly manipulate a RAP whereas a rapClient only indirectly effects a RAP through its current rapServer. Any user-written UNIX application may become a rapClient by inheriting from the right C++ class.

4 Sun Platform

4.1 RAP Client-Host Protocol

A rapClient communicates to a rapServer using the RAP Client-Host Protocol which performs both presentation and application level functions. Built on top of TCP/IP, the protocol provides:

- Data type safety: Type check the sequence of data messages sent to the rapServer. The rapServer will discard a RAP message if it does not strictly follow the format.
- Floating-point number conversion: Floating-point numbers on the rapClient end of the connection must be in IEEE[IEE85] format, but on the server end must be in in TMS320C30 format. A rapClient is assured that floating-point numbers sent to a rapServer in IEEE format will be converted to TMS320C30, and numbers returned from the TMS320C30 will be converted to IEEE.
- Virtual RAP node numbers seen by the rapClient: The rapClient will only see node numbers 0 through $(numberOfNodes - 1)$ regardless of the extant physical node numbers.
- A small amount of user security. The current user's job may not be disturbed by any other user.

Similar to an RPC, a rapClient uses function calls to send protocol messages to its rapServer and receive return values. The rapServer interprets messages and executes the corresponding action on the RAP machine. The rapServer maintains a current (board ID,node ID) tuple which will apply to all future rapClient protocol messages. The rapClient may change the current tuple by sending a message to change virtual node numbers; the rapServer will translate the virtual node number into a hardware (board ID,node ID) tuple (if the virtual node number is -1, future protocol requests will affect all nodes). A rapClient protocol message to a rapServer consists of a message number followed by some number of parameters. Parameters may be of type int, float, or char*. Each protocol message returns a value that may signal an error. Optionally, one additional word value may be returned from the rapServer. Large arrays of data may be returned to a rapClient using typed data messages.

One include file is used to define the protocol for all protocol implementors. This is done by defining a C macro to extract the needed information out of macro calls defined in an include file. The include file contains macro calls which completely define each protocol message; information is given by the actual parameters. Each definition contains information that will supply all necessary information about the protocol message. The C macro can be defined differently each time the file is included by using formal parameters in different ways. Thus different information is extracted at each include instance. The information about the protocol, however, is contained in one place, so if any protocol changes are made, only a recompile is necessary to update all protocol implementors.

In the include file protocol definition, C macros are called as:

```
RAPPROTO(cmdNam,cmdNum,cmdSym,fprms,a1,a2,a3,a4,pm1,pm2,pm3,pm4,rv,r)
```

where `cmdNam` is the name of the protocol command in string form; `cmdNum` is a unique command number uniquely determining the command at run time; `cmdSym` is a case sensitive unique command symbol which can be used to create a routine name to call; `fprms` is the ANSI C style procedure header. The types correspond to the parameters `pm1,pm2,pm3,pm4`; `a1,a2,a3,a4` represent the types of `pm1` through `pm4` and must be the symbols `s`, `f`, or `d` for string, float, or decimal integer respectively; `pm1,pm2,pm3,pm4` are the string types of the parameters (``s'' for arbitrary string, ``f'' for a floating-point value, ``d'' for a 4 byte integer, or `0` for no parameter sent); `rv` is the type of the return value of the function (``d'' to return an integer value, or `0` for no return value); and `r` is the symbol form of the return value (`d` if for 4 byte integer return code, or nothing if no return value).

A typical protocol definition in the include file looks like:

```
/* specify user name, user id, return socket number, */
/* and requesting host name */
RAPPROTO("user",2,user,
    (char *p1,int p2,int p3,char *p4,int *rp),
    s,d,d,s,
    "s","d","d","s", /* user, uid, return socket, hostname */
        "d",d)    /* return value is the number of active nodes */
```

In this case, the first parameter, `p1`, is the textual user name. The second is an integer user ID. The third is the return socket number; the host containing a RAP will connect back using this socket for asynchronous output. The fourth is the connecting host name (this is necessary since VxWorks does not have the `gethostbyaddr()` system call).

Some typical protocol requests are:

`user` – Tell the `rapServer` the identity of the `rapClient`.
`load` – Load a TMS320C30 executable file onto a RAP node.
`run` – Run a program.
`peek` – Peek at the given address on a RAP node.
`modify` – Poke a value at the given address on RAP node.

A typical application would use the protocol definitions as follows:

```
#define RAPPROTO(cmdNam,cmdNum,cmdSym,fprms,a1,a2,a3,a4,pm1,pm2,pm3,pm4,rv,r)\
    extern rapReturnType rapProto # cmdNam fprms ;
#include "rapClientHostProto.h"
#undef RAPPROTO

/* use the defined external definitions */
...
```

4.2 UNIX RAP Client: C++ RAP Interface

The `rapClient` interface provides flexibility to users writing RAP applications and makes it relatively easy to incorporate the RAP as a computational server.

A `rapClient` lives on the UNIX side of the connection with a `rapServer`. It is written in C++ but a C interface will be very easy to provide. User applications running on a Sun which

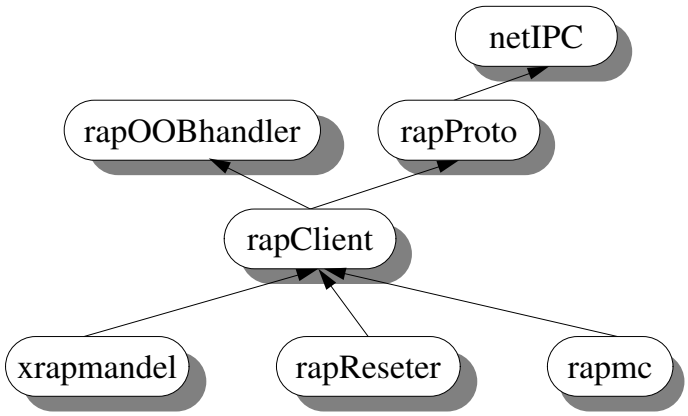


Figure 2: RAP Client class hierarchy

are written in C++ may inherit from the C++ `rapClient` class and may then send protocol messages to the RAP by calling `rapClient` member functions. An application can thus use the RAP as a computational server for an interactive graphics application (although no form of real-time is guaranteed) or for a UNIX filter (some waveform display programs like the Entropic Signal Processing System (ESPS) [Ent] use UNIX filters as a means of adding custom behavior).

A `rapClient` references nodes using virtual node numbers. A virtual node number ranges from 0 to $4 \times \text{numBoards} - 1$ and is translated by the `rapServer` to a hardware (board ID, node ID) tuple. There will be many boards with different hardware board ID and node ID values[Bec90]. With virtual node numbers, any configuration of hardware board ID and node ID RAP boards may be installed in any order, and the user will only see consecutive nodes numbers starting at zero.

4.2.1 C++ `rapClient` Classes

The `rapClient` class hierarchy is shown in figure 2. The classes and `rapClient` applications are described below.

`netipc` Contains the low level internet domain (AF_INET) socket interface. Constructing a `netipc` takes a host name as an argument and connects to the port given by RAPPORT which a RAP server listens on. `netipc` supports typed data messages (int, float, string) which are used to implement the RAP Client/Host Protocol. This ensures that all senders and receivers only use known message types and lengths.

`rapOutOfBand` Supports outgoing RAP data (from the RAP to the `rapClient`) and sets up a UNIX SIGIO signal handler to catch RAP standard and error output that occurs asynchronously with respect to a `rapClient`. The signal handler is invoked whenever there is activity on the socket coming from the `rapServer`. It will read the message and write the data via file descriptors set up by the `rapClient`. The default RAP stdout and stderr will go to file descriptor 1 and 2 respectively. A `rapClient`, however, can set these file descriptors to reference a pipe or socket which will redirect RAP output to a file or pipe RAP output to other UNIX processes. `rapOutOfBand` also supports asynchronous messages coming from the RAP which are sent to all active `rapClients`. Asynchronous messages include status (the RAP program has finished, the RAP is ready to run, and the RAP terminated with an error),

standard I/O messages, and "typed data received" messages (using integer or floating-point data).

`rapProto` This object, a child of `netipc`, contains member functions that implement RAP protocol messages. It also contains a member function `rapProto::sayHello()` that will send the required initial set of protocol messages (using `hello()`, `user()`, `ready()` and the `rapOutOfBand` object) to the `rapServer`. The `sayHello()` routine provides the standard way all `rapClient`s must initiate a dialog with the RAP. It will prompt the user if the newly activated `rapClient` is not in the front of the user queue (to be described in the `rapServer` section) and will automatically continue when the `rapClient` reaches the front of the queue (batch jobs are implemented by having a `rapClient` then continue to send RAP messages). `sayHello()` also notifies the user if an error occurred while connecting to the RAP.

The `sayHello()` function performs:

```
// make sure connection is working by
// sending the Hello protocol request.
hello();

// tell rapServer information about myself using user()
user(myUserName,myUserId,myConnectPort,myHostName,
    &numActiveNodes);

// tell rapOutOfBand object to accept the return connection
acceptConnection();

while (!ready()) {
  if (weAreInRapUserQue)
     // give the user the option to look at the queue. Leave
     // or continue waiting.
     ...
  else if (anErrorOccurred)
     // do correct error processing and return
     ...
}
```

`rapProto` uses the RAP Client/Host Protocol include file to construct its own member functions which use `netipc` to communicate with the RAP. Changes in the protocol definition header file will require minimal or no changes in the `rapClient` source.

Some additional member functions are:

- load(char *fileName);
- run(char *arguments);
- peek(unsigned rapAddress, char *symbolName, int *result);
- modify(unsigned rapAddress, char *symbolName, value);
- input(char *string);

`rapClient` This object encapsulates the three preceding objects by inheriting from both `rapProto` and `rapOutOfBand`. It redefines the virtual member functions for "typed data received" messages. Descendents of `rapClient` may further redefine them causing different actions for incoming data. There are also virtual member functions for RAP standard and error output. The default virtual functions for the "typed data received" messages ignore the incoming data and the default virtual functions for RAP standard error and output will send the data to the UNIX process's stderr or stdout respectively. `rapClient` also:

- Contains routines to query the state of each RAP node.
- Sets up default signal handlers for SIGINT, SIGSTOP, and SIGHUP.
- Contains member functions which can manipulate the file descriptors that the `rapOutOfBand` object uses.

A single user may create multiple `rapClient`s in different UNIX processes. Thus, a user may have multiple `rapClient`s being served by one RAP machine.

4.2.2 Simple RAP Client

The following figure is an example of a simple `rapClient`:

```
// A C++ program to reset all RAP nodes on
// the machine <raphost.berkeley.edu>
#include <rapClient.h>

class rapReseter : public RapClient {

     public:

        // when constructed, the rapReseter object will
        // reset all nodes of the RAP Machine it is connected to.
        rapReseter(char *host)
            : RapClient(host)  // pass host on to rapClient
        {
                // tell the rapServer that further commands
                // should effect all nodes.
                nodeset(GLOBALRAPNODE);
                // reset the current set of nodes.
                reset();
        }
};

main(int argc,char *argv[]) {
    // tell the rapReseter object rr to connect to the host 'raphost'
    rapReseter rr("raphost.berkeley.edu");
}
```

4.2.3 RAPMC: Complex RAP Client

RAPMC is a `rapClient` that is used for interactive controlling and debugging of a RAP machine. By sending sets of RAP Client/Host Protocol requests for each command entered at the terminal, RAPMC adds interactive control of the RAP directly from the RAPMC command line. It also gives users the ability to change the destination of RAP standard and error output. Users may start up multiple RAPMCs and have different RAP nodes send output to each RAPMC.
RAPMC supports interactive commands to:

- Load and Run RAP programs.
- Reset RAP nodes.
- Display the RAP user queue.
- disassemble TMS320C30 instructions.
- Examine or modify RAP Memory.
- Redirect RAP node output to a file or UNIX process.
- Execute Command Scripts.
- Send ASCII text to a RAP node's standard input (stdin).
- Wait for RAP jobs to complete.

A typical command script for RAPMC follows:

```
0> node *
*> 2 > yo.out1
*> 3 | egrep -i error
*> load yo
*> run foo
*> wait
*> 2examine i 0x2c
*> reset
```

See [KB90] for a complete user's reference on RAPMC.

4.2.4 Mandelbrot: Graphical RAP Client

The RAP Mandelbrot program is an example of using the RAP as a computational server for an interactive graphical UNIX application. A subclass of `rapClient`, `rapMandel` uses the RAP for the computational portion of the Mandelbrot/Julia set fractal generation algorithm. It uses the "typed data received" messages (that `rapClient` supplies as virtual member functions) to receive data that is used as indices into the color map for a Sun workstation. `rapMandel` redefines the virtual routine "rapClient::intsReceived()" to directly call an X11 library function. Therefore, asynchronous events (caused by the RAP) perform the drawing while the synchronous portion of the program coordinates the communication.

Benchmarks for a screensize of 550x590 pixels, 8 bits per pixel, and a maximum of 256 inner loops per pixel follow:

	SPARC 1+ Single Prec.	1 Board RAP	2 Board RAP	3 Board RAP
Average Run	46.7 sec	8.7 sec VxW 4.4 sec Sun	8.7 sec VxW	3.75 sec Sun
All Points in Set	4:02.9 min	19.2 sec VxW 15.9 sec Sun	11 sec VxW	6.5 sec Sun

The `rapServer` running on a Sun or under VxWorks (to be described later) is marked with "Sun" or "VxW" respectively. Right now, I/O is the limiting factor in the average run case as can be seen by noting that there is no performance improvement in going from 1 to 2 boards with the VxWorks system and a small performance improvement in going from 1 to 3 boards with the Sun system.

It was expected that the RAP I/O in a Sun Workstation would be faster since VxWorks has networking problems. This appears to be the case as can be seen from the table. But, as more boards are added, I/O should be a limiting factor since we will have reached ethernet bandwidth. We plan, however, to implement a `rapClient` interface identical to the current one but which directly manipulates a RAP contained in the local machine. This will allow existing `rapClient`s to run without change by re-inheriting.

Apart from being a visually pleasing demonstration of the RAP's performance, the Mandelbrot/Julia Set example shows that RAP works well on a problem outside the domain of artificial neural network algorithms.

4.3 RAP Server: Sun Workstation

`rapd` is a program which runs on a Sun UNIX workstation and that accepts connections from `rapClient` processes. It receives RAP Client/Host protocol messages from `rapClient`s, and then directly manipulates the RAP machine in the workstation. Written in C++, `rapd` is started as a daemon that runs on a workstation. It communicates to the RAP through a UNIX device driver (accessed by one of the `/dev/rap?` device files).

A user may wish to have more than one RAP system in a Sun. If so, more than one version of `rapd` may run with each operating on different `/dev/rap` files. A `rapClient` will decide to which `rapd` it should connect.

`rapd` maintains the RAP user queue which holds the set of users waiting for the RAP. The first user in the queue is called the active user. If this user creates additional `rapClient`s, they will immediately be given RAP access. If a different user starts up a `rapClient`, he/she will be queued until the active user's `rapClient`s are finished. At that point, the next user in the user queue will become active. A mechanism for batch jobs is thus provided. A user on a Sun may start a `rapClient` in the background. When `rapd` causes that `rapClient` to become active, it will start running. If the user sets the `rapClient` to finish and exit when it is done, the next user in the user queue will get RAP access. `rapd` also provides a way for one user to start up multiple `rapClient`s. Each `rapClient` may manipulate a different RAP node (which is useful for a MIMD style of programming), or receive output from a different node. One `rapClient` might be graphical output, whereas another may just be a textual interface. If a RAP job is started on a workstation, one may later log in from a dial up line and check progress without disturbing the current job.

The following is a description of the `rapd` internal objects shown in figure 3:

que,queObject,slList,dlList,generator,intHasher A queue, objects for the queue, singly and doubly linked list objects, and an abstract generator class.

bridge Each `rapClient` has its own instance of a `bridge` object that will keep state for it, maintain its connection, and communicate to the RAP boards through the `rapSystem` object. When all active `bridge`s of the current user finish, the next set of `bridge`s in the user queue become active and their jobs start executing.

Bridges may be in one of five states.

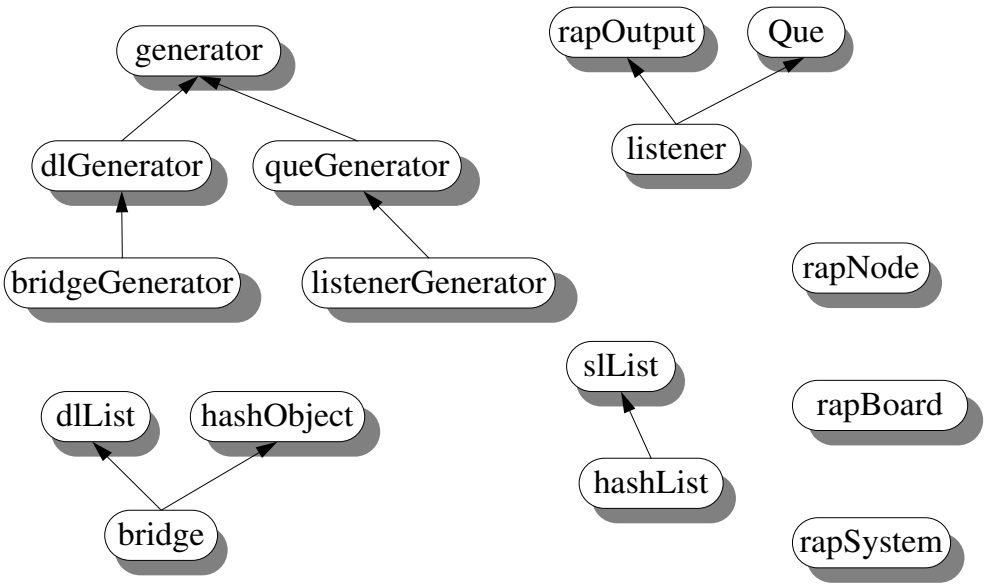


Figure 3: `rapd` class hierarchy

inchoate Until the `rapClient` has sent his user name, user ID, host name, and return socket number to `rapd` (using the `hello()` and `user()` protocol requests), a newly created `bridge` will be in the inchoate state where it is unable to manipulate the RAP or communicate to any user. Once it has received the necessary information, a `bridge` will set itself to the waiting state.

waiting If the `bridge`'s user is not the active user in the user queue, the `bridge` will remain in the waiting state until it reaches the front. In this state, a `bridge` may only query its own state or send a listing of the user queue to its `rapClient` (so that the `rapClient` may decided to exit if the queue is too long).

active In this state, a `bridge` is currently receiving commands from or sending output to its `rapClient`. A `rapClient`'s `bridge` is at the front of the user queue and currently has control of the RAP.

dead After the `rapClient` has sent a `quit` protocol message, the `bridge` will place itself in this state. Any `bridge`s found in this state will be deleted.

ghost After the `rapClient` has sent a `shelve` protocol message, if its `bridge` is the last remaining `bridge` for the active user, it will set itself to `ghost` state. This will cause the `bridge` to serve as a place holder for the user at the front of the queue. When a new `rapClient` of the same user who sent the `shelve` message connects to `rapd`, that user will get RAP access.

kill If a user sends a kill protocol request, the bridge will set itself to the `kill` state. This will cause all bridges of the current user to be eliminated from the user queue and the current RAP job (if any) to be destroyed.

listener The `listener` object monitors all sockets and file descriptors using the `select()` UNIX system call and dispatches all work to the appropriate `bridge`s. It listens for activity on all

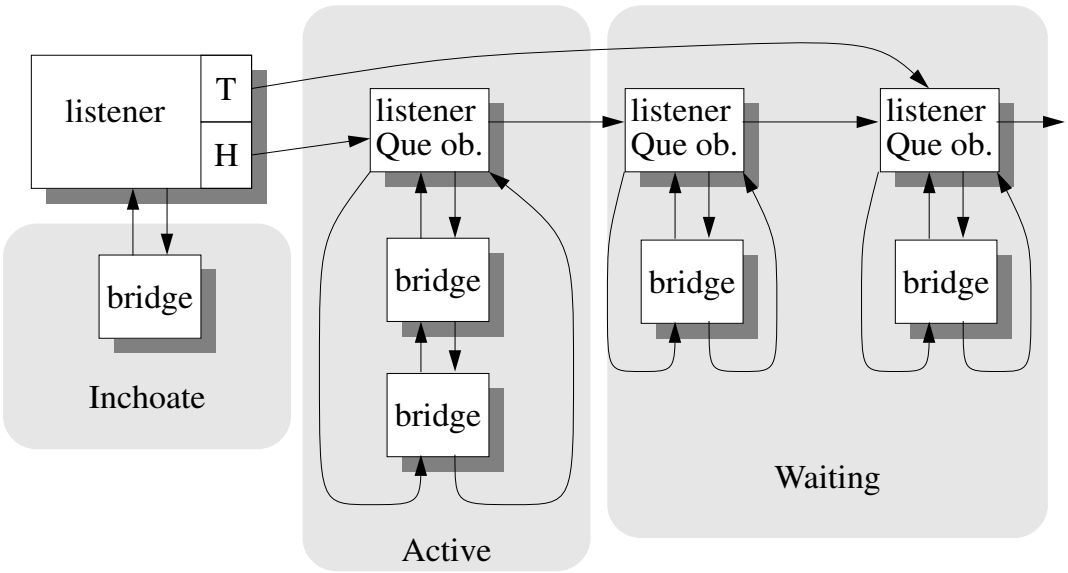


Figure 4: Listener Data Structure

`bridge`'s incoming message sockets, on all file descriptors corresponding to the `/dev/rap` files, and on the socket for incoming connections from new `rapClient`s. When a file descriptor becomes active, the `listener` discovers who it belongs to, and asks the appropriate module to process it.

The `listener` object is also a queue of non `inchoate` bridges (the user queue) and maintains a list of `inchoate bridges` (see figure 4). When a new `rapClient` connects to `rapd`, the `listener` will create an `inchoate` bridge that corresponds to the new `rapClient` and will insert the `bridge` in the `inchoate` bridge list. As new messages come in for the `bridge`, the requests are hashed, keyed on the file descriptor, to obtain a `bridge`. An `inchoate bridge` may, upon receiving the necessary information, change its state to `waiting`. The `listener` will then place the `bridge` into the user queue based on its username and userid number. The bridge will then become either active (if the username and userid is the same as the active user) or will remain waiting in the correct queue location.

rapNode `rapNode` objects correspond directly to and provide class member functions that manipulate physical RAP nodes.

The `rapNode` object:

- Maps all RAP node physical memory into user virtual memory using the `mmap()` UNIX system call.
- Keeps track of the state (run or reset) of the physical RAP node.
- Manipulates all RAP node registers.
- Supplies a public `int& operator[](int nodeAddr)` function so a `rapNode` can be used as an array of RAP memory indexed using node addresses.
- Keeps track of the state and style of all open files for the RAP node.
- Keeps a current working directory for the RAP node.

- Stores the arguments of the previous run so that they may be used for a run given without arguments.
- Stores a copy of the on-chip memory section and `.cinit` section (initialized C data) of the previously loaded file.
- Before a run, copies the stored arguments and on-chip memory to the beginning of the RAP node stack. Since TMS320C30 on-chip memory is not addressable from the VME bus [Bec90], the data must be copied from the stack to real TMS320C30 on-chip memory by the DSP boot code. `rapNode` also copies a fresh version of the `.cinit` section to RAP memory.
- Keeps track of and uses node communication addresses for sending and receiving data transfer requests from the DSP.
- Keeps track of the running time of the RAP node.
- Supplies routines which check the state of an exit loop that might be running on the RAP. If the loop is running, the RAP node has successfully completed its job and it is safe to re-run the program without reloading. This avoids unnecessary hard resets that will corrupt memory.
- Processes RAP node interrupts caused by RAP system calls and break points. The lowest level RAP system calls are:

  `exit()` – exit the program
  `open()` – open a file
  `close()` – close a file
  `creat()` – create a file
  `read()` – read data from a file
  `write()` – write data to a file
  `lseek()` – seek to position in the file
  `cd()` – change current working directory
  `rapClientWriteInts()` – direct integer write to a `rapClient`
  `rapClientWriteFloats()` – direct floating-point write to a `rapClient`

- Keeps a queue of data buffers which the RAP node may read as standard input.
- Supplies public class member functions to:
  - reset and set the node
  - load the node with a TMS320C30 executable file
  - run the program
  - return the current working directory for the node

rapBoard The `rapBoard` class sets up the communication with a RAP using a RAP device file that provides access to the RAP device driver. The device file is specified by `/dev/rap?` where `?` is the hardware RAP board ID and the minor device number (the minor device number and hardware board ID must be the same). Each `rapBoard` instance corresponds to a physical RAP board contained in the local machine. When constructed, the `rapBoard` instance must be given a RAP device file. Successfully creating an instance of a `rapBoard` signifies that a physical RAP board was found. Subsequent references to the physical RAP board can be made through the `rapBoard` object and any one of the four `rapNode` objects which compose `rapBoard`.

The rapBoard class provides:

VME Slot 2	VME Slot 3	VME Slot 4
RAP BOARD 1	RAP BOARD 0	RAP BOARD 2
/dev/rap1	/dev/rap0	/dev/rap2

Figure 5: RAP Boards in a VME Bus

- A `rapNode& rapBoard::operator[](int nodeNum);` member function to reference a certain `rapNode` object. If, for example, `rb` is an instance of a `rapBoard` class, `rb[0][0x300]` will be RAP node word address `0x300` on `rapNode` zero on this board.
- Public memory and ring Xilinx programming routines.
- Physical to virtual memory mapping of all RAP board and node registers using the mmap() UNIX system call.
- The following public routines:

  `reset()` – Reset all nodes of this board.
  `set()` – Sets all nodes of this board (release the reset bit).
  `load()` – Load a TMS320C30 executable file into all nodes.
  `run()` – Run a program on all nodes of this board.
  `pwd()` – Print the current directory of all nodes to the active `rapClient`s.

rapSystem The `rapSystem` class groups together `rapBoard`s and does the translation of `rapClient` virtual node numbers to physical (board ID,node ID) tuples. At startup, an attempt is made to construct a `rapBoard` object using each RAP device file given in the `rapSystem` constructor's arguments. If a `rapBoard` object is successfully created, a software board ID number is incremented (which is independent of the hardware board ID), and the next RAP device file is tried. Physical board ID's do not need to be equal to virtual board IDs. For example, suppose we have three RAP boards in a system with physical board IDs one, zero, and two, and corresponding RAP device files `/dev/rap1`, `/dev/rap0`, `/dev/rap2` but the boards are out of VME slot order (shown in figure 5).

If a rapSystem is constructed as:

```
char *raps[] = { "/dev/rap1","/dev/rap0","/dev/rap2" };
rapSystem myRapSystem(3,raps,myListener);
```

then virtual board ID 0, 1, and 2 will refer to hardware boards 1, 0, and 2 respectively. Additionally, virtual node numbers 0-3, 4-7, and 8-11 will refer to physical RAP boards 1, 0, and 2 respectively. Of course, this will only work correctly if the ring connectors on the RAP boards are such that physical board 1's `ring_put` writes to physical board 0, and physical board 1's `ring_get` reads from physical board 2 (and physical boards 2 and 3 are correspondingly connected correctly).

The `rapSystem` class contains the load routine which both `rapBoard` and `rapNode` use to load TMS320C30 executable files. The load routine is contained here so a global load (to all nodes of all boards) will read the file only once from disk.

The `rapSystem` class provides the following routines:

`numBoards()` and `numNodes()` return the number of successfully created `rapBoard` and `rapNode` objects respectively.

RAP interrupt handler routines to handle interrupts from RAP boards. When there is activity on a RAP board's file descriptor, the `listener` object will notify the `rapSystem`.

`reset()`, `set()`, `run()`, `load()`, `chdir()`, `storeInputText()`, and `pwd()` which effect or pertain to all nodes of all boards. Therefore, a user may reset all nodes of all boards using the `rapSystem` object.

`reset()`, `set()`, `run()`, `load()`, `peek()`, `poke()`, `chdir()`, `storeInputText()`, and `pwd()` which are similar to the above but affect or pertain to only a given virtual node. For example:

```
rapSystem::load(char *fileName)
```

will load the TMS320C30 file to all nodes of all boards but

```
rapSystem::load(int nodeNum, char *fileName)
```

will only load to the RAP node that corresponds to the given virtual node number.

rapOutput A abstract `rapObject` class provides `printf()` style routines that `rapNode`, `rapBoard`, and `rapSystem` use for directing RAP output and system messages. `listener` inherits from this class and determines the correct `rapClient` destination.

4.4 Device Driver Design

A UNIX device driver interfaces with `rapd` and provides direct hardware control. Two different device drivers exist, `rapmem` and `rap`. A user wishing to install several RAP boards in a Sun will install one of these device drivers by editing the necessary kernel configuration files and rebuilding and installing a new UNIX kernel [BBK91]. A user should name the RAP devices files consecutively from `/dev/rap0` to `/dev/rapn` where n+1 is the number of RAP boards in the Sun. These files should correspond directly to RAP boards with physical board IDs ranging from 0 to n. *The Minor device number of the device file MUST correspond to the hardware RAP board ID for memory addressing to work correctly.*

rapmem This device driver supports RAP memory mapping using the `mmap()` UNIX system call. It was done primarily to test out accessing RAP memory from a user process and to test out the kernel physical memory to user virtual memory routines supplied by the kernel. Users will normally create files called `/dev/rapmem0` through `/dev/rapmemn` for RAP boards with RAP board id's 0 through n.

rap This device driver supports `open()`, `close()`, and `mmap()`, like rapmem but adds `select()`, `ioctl()` and a RAP interrupt handler. RAP interrupts may be detected by a user process in one of three ways (selectable using the `ioctl()` system call):

1. Poll for an occurrence of an interrupt. A user process may then proceed to check which node caused the interrupt by looking at node memory. Polling is done by calling `select()` with the `struct timeval` timeout actual parameter pointing to a zero value.

2. Sleep until a RAP interrupt occurs. Once a RAP interrupt occurs, the user process will wake up and may then check which node caused the interrupt. Sleeping is induced by calling `select()` with the `struct timeval` timeout actual parameter either pointing to the time-out value or equal to NULL which will cause `select()` to block indefinitely.
3. Use asynchronous signals. If this mode is selected, a RAP interrupt causes a SIGIO signal to be sent to the user process that has the device open. Asynchronous interrupts are activated with the `ioctl()` system call using the RAPIOASYNC option.

Although the RAP is a two address space board (it uses both VME 16 bit and 32 bit address spaces), it was not necessary to map any RAP memory into the kernel virtual address space. RAP memory is mapped (by `rapd`) into user process virtual address space. Only the RAP board and node registers are mapped into kernel virtual address space to initialize the RAP at boot time. Thus, during the Sun boot sequence, RAP memory can not be tested by the kernel `rapprobe()` routine. It might seem like this would be an easy task. However, since the Xilinx memory part has not yet been downloaded at Sun boot time (so RAP memory does not yet exist), and since there is no easy way to read disk files from the kernel nor did it seem reasonable to keep a copy of the Xilinx program file in the kernel, all memory testing is deferred until user process time.

5 VxWorks/Heurikon Platform

The initial RAP host consisted of the VxWorks operating system[Win] running on a Heurkon[Heu] MC68020 board.

5.1 RAP Server: VxWorks/Heurikon

VxWorks, supplied by Wind River Systems, is a real-time operating system. It is real-time in that it uses priority based preemptive scheduling; the highest priority runnable process always runs. If there is more than one runnable processes with the highest priority, round-robin time-slicing is used.

In our configuration, VxWorks runs on a MC68020 based CPU board (a Heurikon HKV2FA) that is contained (along with an ethernet board) in a 9U 21 slot Dawn VME card cage. The initial RAP host software was developed under VxWorks for several reasons.

- VxWorks contains a debugger with which tasks may be controlled and monitored.
- There is no virtual memory support on the Heurikon board; thus one non-trivial complexity is eliminated. Debugging a new board is easier since all addresses are physical rather than virtual. We used the VxWorks debugger to directly examine RAP memory at VME bus addresses without having to modify page or segment tables in an MMU (which would have been necessary using a Sun).
- The card cage we use has physical advantages in that we can attach extender debugging boards for use with a logic analyzer.
- It was originally thought that real-time OS support was necessary since the primary RAP application is speech recognition. It turns out, however, that this can be more efficiently implemented using the serial ports on the TMS320C30 (for which an interface exists on the RAP boards) as a real-time interface to other boards such as A/D and D/A converters. Thus none of the real-time features of VxWorks were used.

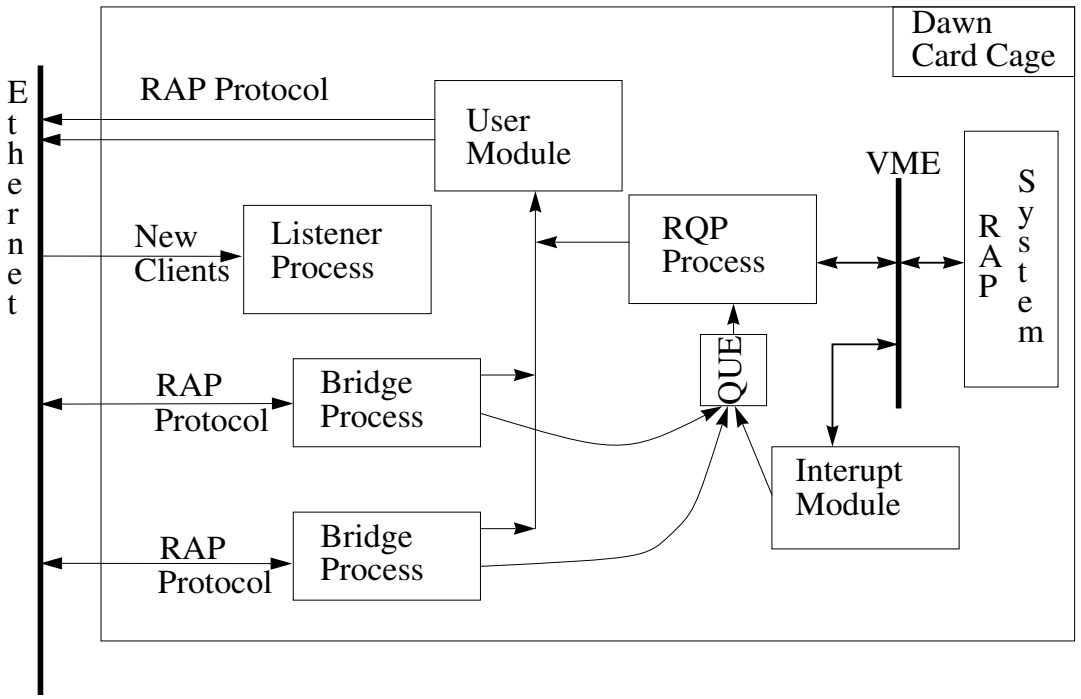


Figure 6: The VxWorks RAP Host Server

- It was thought that the more popular RAP configuration would be the VxWorks/Heurikon system. It now appears the RAP in a Sun Workstation will prevail.

The VxWorks RAP server, written in C, uses similar modules to the Sun server `rapd`. Each VxWorks module, however, is a process rather than an instance of a class. This was done since VxWorks processes are much more lightweight than UNIX processes and since debugging was facilitated by separating the modules into more distinct entities. However, each process still shares memory. VxWorks does not provide a separate address space per process.

The VxWorks server implements the RAP Client/Host protocol. Again, like the Sun server, multiple `rapClient`s may connect to the server. Also a user queue is maintained which allows multiple `rapClient`s of the same user to simultaneously manipulate the RAP. `rapClient`s of differing users are queued. The VxWorks server is depicted in figure 6.

VxWorks modules include:

Listener The listener module monitors incoming requests for new `rapClient` connections. When it gets a new request, it spawns a bridge task to handle further incoming requests from the new `rapClient`.

User Queue Module This module maintains an ordered queue of users. The second user in the queue will get RAP control after the first user is done processing and has exited. The user queue module also keeps track of where output for each of the `rapClient`s should go by supplying a set of I/O routines such as:

```
rapClientStdOut(int nodeId,char *message);
rapClientErrOut(int nodeId,char *message);
```

These will send a textual message to all current `rapClient`s. A lower level of output is provided with:

```
usrNodeOutput(int outputType,int nodeNum,char *buff,int bufflen);
```

which will send any general message of type `outputType`, to all users. `outputType` distinguishes between standard output, error, or system messages; `nodeNum` determines the source node; and `buff`, along with its length `bufflen`, provides the data.

Bridge Every `rapClient` has an associated bridge. A bridge manages the communication between each `rapClient`, the user queue, and RQP. It does this by passing requests (using the RAP Client/Server Protocol) sent from a `rapClient` to the various modules.

Interrupt routines The interrupt routine is invoked by an interrupt from the RAP. When one occurs, the interrupt routine obtains the necessary information from the node requesting an interrupt and adds an entry to the queue read by RQP.

Request Queue Processor: RQP All serialization of commands from the various bridges and the interrupt routine is done with a globally shared semaphore-protected queue of which RQP is the only reader. RQP takes entries off this queue and manipulates the RAP. Commands such as `load(int nodeNum, char *fileName)` or `run(int nodeNum, char *args)` from a `rapClient` are placed into the queue by a bridge. RAP system calls such as `read(int fileDescriptor,char *buff,int len)` are placed into the queue from the RAP interrupt routine. RQP thus serializes the parallel components of the system and reduces the chance of race conditions. The processing of events from the parallel execution of multiple DSP's in a RAP board and multiple `rapClient`s on machines in a heterogeneous network is guaranteed to be serialized using this queue scheme.

RAP node requests include:

`read()`: read from file
`write()`: write to file
`lseek()`: seek position
`open()`: open new file
`close()`: close file
routines to write data directly to a `rapClient`

`bridge` requests include:

Load a file to a node
Run a program
Read a RAP node memory location
Write a RAP node memory location
Reset a RAP node.

Several problems encountered while using the VxWorks/Heurikon system include:

- There is no memory protection so it is easy to corrupt VxWorks.
- Several non-trivial VxWorks bugs were discovered including problems with NFS and process creation.
- The VxWorks process stack is fixed and network routines often caused a stack overflow.

Thus, we recommend using the Sun platform for any software development research involving the RAP.

6 TMS320C30 Platform

The platform where RAP code runs is composed primarily of TMS320C30 DSP chips [Tex88a] [Tex88b] [Tex89]. In order to run a new application on the RAP, a user must first explicitly parallelize an algorithm for the RAP architecture. We have minimized the difficulty in doing this, however, by providing a large assortment of standard functions one would want for connectionist and matrix algorithms and by providing a familiar environment in which to program.

There are two styles of programming: SIMD and MIMD style. In MIMD style, each node of a RAP loads a different executable image and runs different code. The user must make sure that communication between nodes is done properly to avoid deadlock, starvation, and race conditions. Currently, there are several research groups who are interested in using a MIMD style RAP. By having the first node's ring input import data from an external device, each node, in turn, may process data and send it down to the right node for further processing. This pipelined way of processing seems suitable for different kinds of real-time DSP such as digital video compression or audio filters.

In SIMD style, each node contains the same code. Small differences in control flow might occur based on the NODE_ID, but each node performs essentially the same function. This style is ideal for the back-propagation artificial neural net algorithm [RHW86] which is currently being widely used. Additionally, a parallel SIMD Fourier Transform looks potentially viable using the provided distributed matrix routines. The Mandelbrot demonstration and dynamic programming application also use this style.

A user wishing to write RAP code ultimately must be faced with one or more of the following three languages:

- C with the UNIX style C environment.
- TMS320C30 assembly language [Tex88b].
- C++ using AT&T Cfront 2.0.

Most likely, the amount of assembly language programming will be minimal since many of the common functions of the ring are written very efficiently in assembly language. A UNIX like C environment with some extensions to support the ring architecture and to communicate to the host is provided. This makes porting standard UNIX applications relatively easy.

6.1 RAP boot code

Execution on each RAP node begins in the bootstrap code which does the following.

- Initialize the machine configuration (cache, wait states, and page size).
- Copy the .RAM0 and .RAM1 sections from the stack into the on-chip RAM banks.
- Set the uninitialized data areas (.bss section) to zero
- Initialize the stack to special magic words which makes it more obvious in case of a stack overflow.
- Set the chip wait states and bank size.
- Set global variables `N_NODE` (the total number of nodes in this RAP system) and `NODE_ID` (the node ID for this node) based on what the host has passed in.
- Initialize heap memory and memory allocator routines.
- If running in SIMD mode (all nodes are asked to run simultaneously), initialize and sanity check the ring and perform run-time custom compilation.

- Copy `argc` and `argv` onto the stack so `main()` will get them.
- Initialize all registers.
- Call `main()`.
- Call `exit()`.

6.2 UNIX style C programming environment

Many standard C routines are provided in the UNIX style C programming environment. This minimizes the amount of time it takes to port an application to the RAP. A general description of the libraries follow. A complete list and description is given in [KB90].

- Entry to a RAP program starts in the standard C `main()` function. `argv` is obtained from the argument to the `run` protocol request.
- A I/O library exists supporting common UNIX routines like `open()`, `close()`, `lseek()`, `creat()`, `read()`, `write()`, and `cd()`. Additionally, the standard I/O library (stdio) was written for the RAP providing routines such as: `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()`, `feof()`, `printf()`, `fprintf()`, `fscanf()`, `scanf()` and access to the global variables `stdin`, `stdout`, `stderr`
- Memory routines exist including a memory allocator `malloc()` for different memory areas and other routines such as `memcpy()`, `bcopy()`, `bzero()`.
- C string manipulation routines exist such as `strcpy()` and `strcmp()`.
- Several random number generators and a math library including trigonometric and logarithmic functions are provided. Other miscellaneous routines include `panic()`, `setjmp()`, and `longjmp()`.
- Global node constants are provided such as N_NODES (the number of nodes in the RAP system) and NODE_ID (the node number of this node; 0 through $4 \times N_NODES - 1$).
- Ring routines exist that provide access to low level ring communication primitives including a `ring_put()`, `ring_get()`, and `ring_shift()` for both integers and floating-point numbers. Some higher level ring routines include `ring_distribute()` and `ring_write()`.
- Many distributed matrix and vector routines have been provided (and more are being added continuously) so that users may make use of the RAP in a variety of applications. They are "distributed" in that the representation of matrix and vectors may or may not completely be located on one processor at the same time. For example:

  ```
  mul_mv_v(int n_row, int n_col, float *matrix,float *in_vector, float *out_vector);
  ```

  will multiply an `n_row` by `n_col` matrix `matrix` by an `n_col` column vector `in_vector` and produce the resulting vector in `out_vector`.
- Several routines that directly write typed data to the current `rapClient` include:

  - `rapClientWrtInts(int *buffer, int len)`
  - `rapClientWrtFlts(float *buffer, int len)`
  - `rapClientWrtData(unsigned* buffer,int len)`

  The floating-point routine does all necessary conversion between TMS320C30 and IEEE floating-point so that the `rapClient` will get the correct floating-point format. We did not overload the `read()` and `write()` system call. It seemed cleaner to have a different way to communicate typed data to the `rapClient` since the data is treated differently [2]. Overloading `read()` and `write()` would also introduce a difference between the RAP and UNIX versions of those routines.

[2] e.g. in the Mandelbrot program, integer data is used as indices into the color map of an X11 server.

Figure 7 shows a typical SIMD style RAP programming example which will be described in detail. We will see how a typical problem, the forward propagation phase in the back-propagation algorithm [RHW86], is parallelized and how data structures are distributed among the RAP nodes.

As shown in figure 8, each node of the RAP must each have a copy of the input layer. Assuming a 4 node RAP with 8 units per node, node i (varying from 0 to 8) will compute the values for output units $2i$ through $2i+1$ in the output layer. Each node, therefore, needs to maintain only a portion of the weight matrix; node i needing the weights from the input layer to the output units $2i$ through $2i+1$. Once each node has computed its own output units, the `sigmoid()` function will be applied. The resulting output layer, distributed among all of the nodes, is then sent to each other node and a complete representation of the output layer is built which may then be used as the input layer for the next forward propagation step. This process is equivalent to the matrix multiplication operation $Wi = o$ where W is the distributed weight matrix, i is the distributed input vector or input unit layer, and o is the resulting distributed output vector or output unit layer (see figure 9).

6.3 Run-Time Node-Custom Compilation

Several of the lower level assembly language ring manipulation routines described in the previous section often need to make computationally expensive decisions in the inner-most loops. For example, in the `ring_distribute()` routine, each node has an array of values that need to be distributed among all other nodes and each node must ultimately contain an identical large array N_NODE times the size of the original undistributed array. Therefore, if, on each iteration, the `ring_distribute()` routine does one `ring_put()` instruction followed by $N_NODE - 2$ `ring_shift()` instructions (see figure 10), each time `ring_distribute()` gets called, it must do $\lceil arraySize/N_NODE \rceil$ iterations of an inner loop which consists of one `ring_put()` followed by $N_NODE - 2$ `ring_shift()` instructions.

The hard part, however, is keeping track of what element of the array should be written at the top of each inner loop and what element of the array is being shifted in at each `ring_shift()`. Assuming the final array size $farraySize = k * N_NODE$ for some k, at the i^{th} iteration of the outer loop ($0 \leq i < k$), the first `ring_write()` must write element $NODE_ID * farraySize/N_NODE + i$ of the array into the ring. At the j^{th} iteration of the inner loop ($0 \leq j < (N_NODE - 2)$), the return value of `ring_shift()` must be placed in element $((NODE_ID - j + 1) \, mod \, N_NODE) * farraySize/N_NODE + i$ of the array. It is clear then that the routine running on each node is different depending on N_NODE and $NODE_ID$.

Three brute force approaches to implement the above algorithm follow. All of them are unacceptable for the reasons stated.

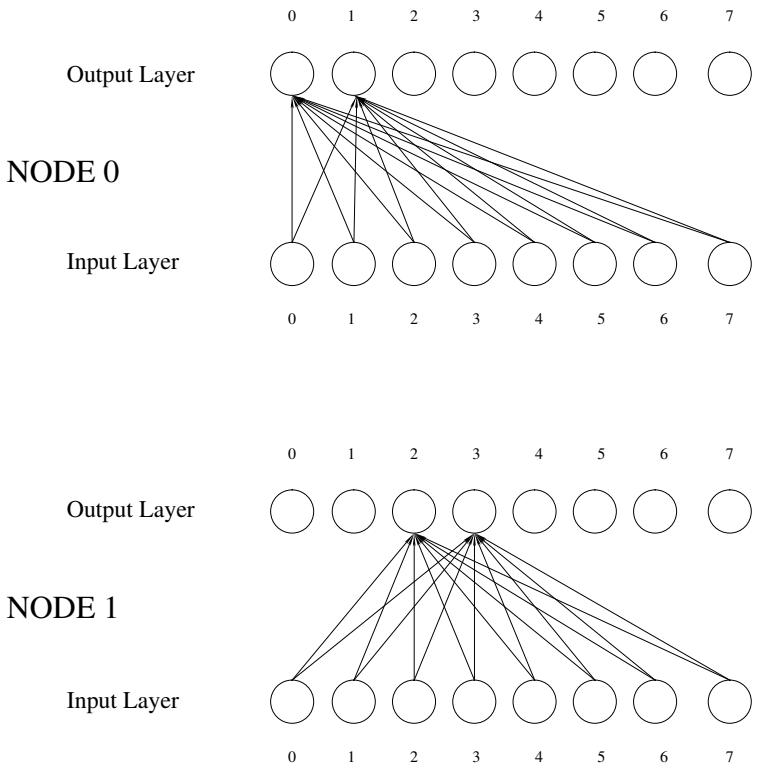
Compute Intensive: Compute the proper array location at each iteration of the inner loop. This involves a large amount of extra integer computation and will slow down the `ring_distribute()` routine appreciably. This is not acceptable since `ring_distribute()` is a frequently called inner routine in most RAP applications.

Space Intensive: Since the above algorithm depends on the variables N_NODE and $NODE_ID$, we can write, compile, and store different `ring_distribute()` routines for all valid combinations of these variables. This will be efficient since no extra run time integer arithmetic needs to be done. Since $N_NODE = 4 * numBoards$ and since $1 \leq numBoards \leq 16$, the number of routines needed is $\sum_{i=1}^{16} 4 * i = 544$ which is far too many routines to keep around.

Indirect: The routine could be implemented by keeping a table of pointers to code stubs and calling each stub in succession. At boot time, the elements of each node's table could be modified to point to the correct stubs for the node. This solution, however, will incur an unnecessary time cost by increasing TMS320C30 pipeline conflicts since it will be necessary to use the address registers to

```
/*
 * Forward propagation step from one layer to the next based on
 * the the "Back Propagation" algorithm
 *
 * The input vector is multipled by a matrix and the resulting
 * vector's elements are put through a sigmoid "squashing" function.
 *
 * The output vector elements are divided among the processing nodes.
 * The matrix rows are also divided so that only the required rows for
 * this node are in memory to compute the node's output vector elements.
 *
 * (in this example, the number of input and output units are equal)
 */

forwardProp(
    /* number of output units on each node of RAP */
    int units_per_node;
    /* complete input_vector (size = units_per_node * N_NODE) */
    float *input_vector;
    /* rows for this node size (units_per_node rows) */
    float *weight_matrix;
    /* complete output vector (size = units_per_node * N_NODE) */
    float *output_vector;
)

{

  int total_units = units_per_node * N_NODE;
  float *this_nodes_output;

  this_nodes_output = output_vector + (units_per_node * NODE_ID);

  /* multiple matrix x vector => vector */
  /* arguments are: # rows, # columns, matrix, in vector, out vector */
  mul_mv_v(units_per_node, total_units, weight_matrix, input_vector,
           this_nodes_output);

  /* sigmoid lookup on output */
  /* arguments are: # elements, input vector, output vector */
  sigmoid_v_v(units_per_node, this_nodes_output, this_nodes_output);

  /* use ring to distribute parts of output vector */
  ring_distribute(units_per_node, this_nodes_output, output_vector);

}
```

Figure 7: RAP forward propagation programming example.

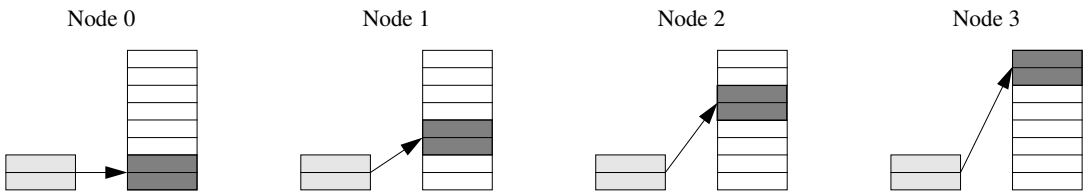


. . .

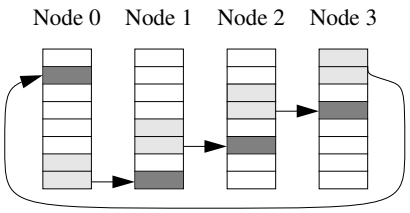
Figure 8: Weight Distribution

$$\begin{pmatrix} \text{node 0} \left\{ \begin{matrix} W_{0,0} & W_{0,1} & \dots & W_{0,7} \\ W_{1,0} & W_{1,1} & \dots & W_{1,7} \end{matrix} \right. \\ \text{node 1} \left\{ \begin{matrix} W_{2,0} & W_{2,1} & \dots & W_{2,7} \\ W_{3,0} & W_{3,1} & \dots & W_{3,7} \end{matrix} \right. \\ \text{node 2} \left\{ \begin{matrix} W_{4,0} & W_{4,1} & \dots & W_{4,7} \\ W_{5,0} & W_{5,1} & \dots & W_{5,7} \end{matrix} \right. \\ \text{node 3} \left\{ \begin{matrix} W_{6,0} & W_{6,1} & \dots & W_{6,7} \\ W_{7,0} & W_{7,1} & \dots & W_{7,7} \end{matrix} \right. \end{pmatrix} \begin{pmatrix} i_0 \\ \\ \vdots \\ \\ i_7 \end{pmatrix} = \begin{pmatrix} \text{node 0} \left\{ \begin{matrix} o_0 \\ o_1 \end{matrix} \right. \\ \text{node 1} \left\{ \begin{matrix} o_2 \\ o_3 \end{matrix} \right. \\ \text{node 2} \left\{ \begin{matrix} o_4 \\ o_5 \end{matrix} \right. \\ \text{node 3} \left\{ \begin{matrix} o_6 \\ o_7 \end{matrix} \right. \end{pmatrix}$$

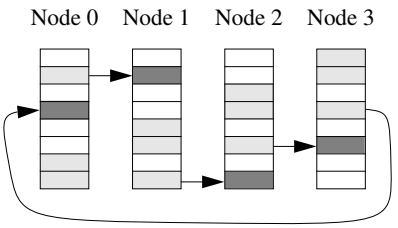
Figure 9: Location of distributed data structures among the various RAP nodes.



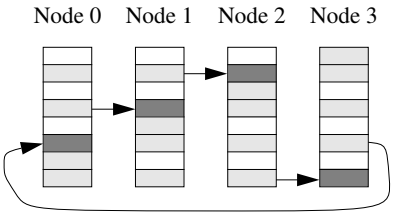
Step 1: Each node has the portion of the array it wishes to distribute. Locations in the actual array are indicated by darker squares.



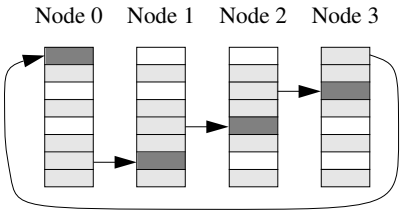
Step 2: After the first ring_write(), each node obtains its left neighbor's first value sent. The new values are indicated by darker squares.



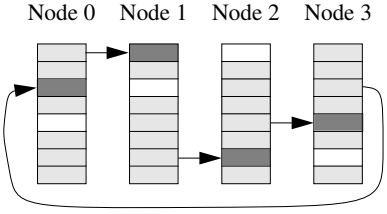
Step 3: After a ring_shift(), the first value written is passed along to the right neighbor who may obtain the value.



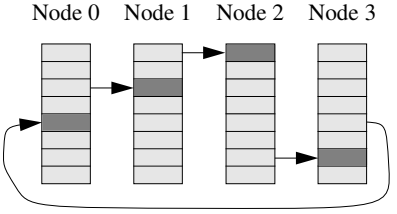
Step 4: After the final ring_shift(), the first value written has been distributed to all nodes.



Step 5: The process continues with each node writting their next value.



Step 6: Again, after a ring_shift()



Step 7: And finally, the entire array is distributed among all processors.

Figure 10: Ring data distribution

```
RC  = # elements in partial vector minus 2 (1 for "pump priming", 1 for RPTB)
AR0 = pointer to input partial vector element to send out
AR1 = pointer to output (full vector) element to be stored next.
      initially this is equal to:
      (output pointer) + (NODE_ID-1)%N_NODE * (# elements in partial vector)
AR2 = address of ring hardware registers
R0  = data coming in from ring
R2  = next vector element value to send out to ring
IR0 = # elements in partial vector
IR1 = (# elements in partial vector) * (N_NODE - 1)
```

Figure 11: Custom compilation register initialization

access the table.

An alternative solution is to compile the correct version of the routine once at run-time. Part of the bootstrap code calls a ring initialization routine, `ring_init()`. Inside `ring_init()`, `malloc()` is called to allocate a buffer for the customized ring code. Then, templates for the instructions in the loop are copied into the buffer. The number and order of these instructions depends on N_NODE and NODE_ID. The only instruction template that must be modified before being copied is the loop size field of the repeat block instruction [Tex88a].

The `ring_distribute()` routine provides a good example of this technique. First, various TMS320C30 registers are setup as shown in figure 11. Then, the first element of the input partial vector is sent out to the ring and the first `ring_shift` is performed into R0. Note that these are done outside the loop to "prime the pump" for parallel instructions in the loop that store the previously shifted data and at the same time shift in new data. Because the first `ring_put` and `ring_shift` are outside the loop, there also has to be the remaining N_NODE-2 shifts and a `ring_get` outside the end of the loop to make a complete last cycle. The inner loop is entered by an indirect jump into the code buffer pointer (called `Ring_distribute_code`) that contains the inner loop customized for that node.

Moving the output pointer AR1 is accomplished by using the post-displacement subtract and modify addressing mode with the step size in IR0 for the (N_NODE-1) backward skips per cycle. The one large forward skip per cycle is done with the post-displacement add and modify addressing mode with IR1. These addressing modes allow the output pointer to be changed without incurring any pipeline delays [KB90].

As an example, lets look at the inner loop for the case of 4 nodes. For node 0 this loop would be:

```
    LDI     *AR0++(1),R2            ; get first invector element

    STI     R2,*+AR2(Ring_put)    ; send invector element to ring
||  LDI     *AR0++(1),R2            ; and get next value to send

;
;   this loop puts a word from invector to the ring
;   it then shifts the ring N_NODE times to get everyones data
;
```

```
    LDI     *+AR2(Ring_shift),R0    ; do ring shift

;***** beginning of inner loop *****
    RPTB    end

    LDI     *+AR2(Ring_shift),R1    ; store previous shift data and do another
||  STI     R0,*AR1--(IR1)

    LDI     *+AR2(Ring_shift),R0    ; store previous shift data and do another
||  STI     R1,*AR1--(IR1)

    LDI     *+AR2(Ring_get),R1     ; what we sent out coming back around ring
||  STI     R0,*AR1--(IR1)

    STI     R2,*+AR2(Ring_put)    ; put causes ext bus to be unusable 2 cycles

    LDI     *AR0++(1),R2            ; get next invector element to put
||  STI     R1,*AR1++(IR0)          ; take big hop forward in output vector

end:
    LDI     *AR1++(1),R3            ; dummy load to increment AR1 (R3 ignored)
||  LDI     *+AR2(Ring_shift),R0    ; shift in data for next cycle
;***** end of inner loop *****

    LDI     *+AR2(Ring_shift),R1
||  STI     R0,*AR1--(IR1)

    LDI     *+AR2(Ring_shift),R0
||  STI     R1,*AR1--(IR1)

    LDI     *+AR2(Ring_get),R1     ; what we sent out coming back around ring
||  STI     R0,*AR1--(IR1)

    STI     R1,*AR1
```

For node 1, however, this loop looks like: (Comments that start with ** indicate instructions that differ from above)

```
    LDI     *AR0++(1),R2            ; get first invector element

    STI     R2,*+AR2(Ring_put)    ; send invector element to ring
||  LDI     *AR0++(1),R2            ; and get next value to send

;
;   this loop puts a word from invector to the ring
;   it then shifts the ring N_NODE times to get everyones data
;
```

```
    LDI     *++AR2(Ring_shift),R0    ; do ring shift

;***** beginning of inner loop *****
    RPTB    end

    LDI     *++AR2(Ring_shift),R1
||  STI     R0,*AR1++(IR0)          ;** do big hop in output array

    LDI     *++AR2(Ring_shift),R0    ; store previous shift data and do another
||  STI     R1,*AR1--(IR1)

    LDI     *++AR2(Ring_get),R1     ; what we sent out coming back around ring
||  STI     R0,*AR1--(IR1)

    STI     R2,*++AR2(Ring_put)    ; put causes ext bus to be unusable 2 cycles

    LDI     *AR0++(1),R2            ;** get next invector element to put
||  STI     R1,*AR1--(IR1)          ;** take small hop backward in output array

end:
    LDI     *AR1++(1),R3            ; dummy load to increment AR1 (R3 ignored)
||  LDI     *++AR2(Ring_shift),R0    ; shift in data for next cycle
;***** end of inner loop *****

    LDI     *++AR2(Ring_shift),R1    ;**
||  STI     R0,*AR1++(IR0)          ;** big hop forward in output array

    LDI     *++AR2(Ring_shift),R0
||  STI     R1,*AR1--(IR1)

    LDI     *++AR2(Ring_get),R1     ; what we sent out coming back around ring
||  STI     R0,*AR1--(IR1)

    STI     R1,*AR1
```

In general these loops can be generated for any number of nodes and any node number by changing the number and order of these instructions; no instruction "patching" is required except for the RPTB instruction where the loop size must be adjusted for the number of nodes on the ring. A small C subroutine generates any of these possible loops. Because the code is generated into an allocated data buffer of the correct size, there are none of the dangers involved in patching code generated by the assembler.

This run-time "custom compilation" approach has the efficiency of the space intensive solution, but takes up the space of the compute intensive solution.

7 Existing Applications on the RAP

mlp: `mlp` is a general program for running and training feed-forward back-propagation networks [KB90]. Many parameters exist (specified in a parameter file) which can control the topology of the desired network and adjust variables that affect the way the network operates. This program can be embedded into other RAP or SPARC programs.

yo: `yo` is the first program to ever run on any RAP machine. Similar to the "Hello World" program in C, yo simply prints out the message: "`Yo, Whazzup`".

mandel: `mandel` is the computational end to the xrapmandel program. Since the algorithm is so small, the data fits in the TMS320C30 register set. Therefore, all pipeline delays were avoided since there are no accesses to external memory. Unfortunately, all of the TMS320C30 parallel instructions need to have one operand be an indirect reference (none of which were needed) so no parallel instructions were used. It seems possible to modify the internal algorithm such that it operates on more than one Mandelbrot row at a time which will thus enable us to use parallel instructions and may (if coded cleverly to avoid pipeline delays) increase performance. Mandel does show, however, that we can get reasonable performance from the RAP machine on a problem for which it was not intended.

dynamic programming: The dynamic programming algorithm is used in speech recognition tasks to perform three functions simultaneously: non-linear time alignment of reference words to input speech, word boundary detection, and the classification of the input speech.

The dynamic programming algorithm must be run once for each sentence to be recognized. Given this, a straightforward way to parallelize this task is to give each RAP node a different subset of the total sentences to be recognized. For example, if there are 100 sentences to be recognized and four RAP nodes available, each node would run the dynamic programming algorithm on 25 sentences.

It is expected that this will work well for the batch-style recognition tasks we are currently dealing with. Of course, for a real recognizer, in which there is only one input sentence, this method of parallelizing the algorithm will not work.

C++ (AT&T Cfront 2.0) on the RAP: C++ code is being developed which, when compiled by Cfront to produce C output which in turn is compiled by the TMS320C30 C compiler, will run on the RAP.

8 RAP Programming Cycle

The programming cycle for the RAP is depicted in figure 12.

9 Comments

Memory Semantics Sufficed: An early design of RAP software called for many special 32 bit registers that interface the RAP with its supporting host machine. These included a WORK, RESPONSE, INTERRUPT, ACKNOWLEDGE, and a MESSAGE register. These were primarily the result of the older assumptions that 1) there existed no shared memory between the RAP and the supporting host, 2) that a master dispatcher (needing very fast registers for a dispatch integer)

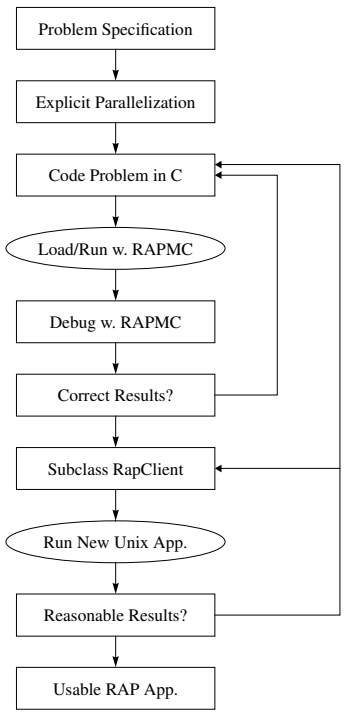


Figure 12: RAP Programming Cycle

would be used to selectively call routines on the RAP, and 3) the application would always use a parent program running on the host.

These registers, however, were eventually reduced down to only 4 bits (DSP and HOST interrupt and acknowledge bits). Most interactions (including all I/O operations) between the RAP and the host were implemented using shared memory. Memory semantics sufficed; we only needed special register semantics for interrupt and acknowledge operations. Also, since this reduced the amount of hardware needed, the entire process was expedited since it is easier and faster to debug software than hardware.

Special Purpose Hardware and General Purpose Applications: The RAP is intended to be a computational server. I/O was not emphasized in the original design goals nor was meant to be one of its talents. General purpose problems, however, didn't do that badly. Both the Mandelbrot demonstration and the dynamic programming have fared well on a machine intended for the back-propagation algorithm.

Object-Oriented Paradigm: The object-oriented programming paradigm worked well and facilitated the design by providing a good abstraction mechanism and a way to reuse software through inheritance.

Portability of C: Although C is predicated to be a portable language, much C code contains the simple assumption that a `char` is really a byte in length and that an `int` is greater than a `char` in length. Any assumptions made like this can cause problems when using C code running on the RAP since the TMS320C30 C compiler produces `char` variables of word length. Strictly following the ANSI C standard, however, should reduce these difficulties. A similar problem arises when C code assumes that the machine is byte rather than word addressable.

RAP and Host Issues: The RAP (TMS320C30) is a word (32 bit) addressable machine while

both host architectures (MC68020 and SPARC processors) are byte (8 bit) addressable machines. The host and the RAP share memory space, however, and this led to several problems. The first one arose while we were using the same C structure declarations as templates over memory for both the RAP and the host. For example, given the following declarations:

```
struct A {
   int a;
   int b;
};

struct C {
   struct A* ap;
   /* ... */
};
```

If we know a `struct C` has been set up by the RAP at a particular RAP memory address, and assign `struct C* cp` to be that address, `cp->ap` will correctly address entry `ap` in the `struct C` but `cp->ap->a` will not correctly address a `struct A`'s `a` integer. This is because `cp->ap` is a RAP address and the host is using it as a host address. Two solutions were used. One defines two C macros:

```
#define HOSTTORAP(haddr) ((unsigned)(haddr)>>2)  /* divide by 4 */
#define RAPTOHOST(raddr) ((unsigned)(raddr)<<2)  /* multiply by 4 */
```

The other has all memory references to the RAP go through a word array with the index be the RAP address. The two correct references to the `struct A`'s `a` integer are:

```
((struct A*)(RAPTOHOST(cp->ap)+rapBaseAddress))->a;
((struct A*)rapMemArray[cp->ap])->a;
```

An additional problem arose when using gcc (the GNU C compiler) to compile the VxWorks MC68020 host with optimization turned on. When copying a character string from the host to a word array on the RAP (since character strings are word arrays for the TMS320C30 C compiler), gcc produced code which byte addressed RAP memory. An assignment of the form:

```
int *iptr;   /* pointer to rap's memory, word address only */
char *cptr;  /* pointer into rap host memory, byte address */
while (...)
  *iptr++ = *(int*)cptr++; /* copy character from host to word on RAP */
```

generates a byte store instruction into RAP memory even though `iptr` is an `int*` and `cptr` is cast to an `int*`. The problem was circumvented by calling a function taking a `char` as an argument and returning it as `int` thus getting around gcc's optimization. A better solution would use MC68020 assembly language.

TMS320C30 pipeline hazard avoidance: While coding for the TMS320C30, we needed to take care not to introduce unnecessary pipeline delays introduced by hazards in the DSP's instruction pipeline. See the guidelines outlined in [KB90].

10 Conclusion

Over the past several months, we have been running real problems on one, two, and three board RAP machines. The RAP machine with its software has provided an easy to use but very fast computational server. Without the RAP, our problems would have taken intolerable amounts of time on the alternative machines we have available (like the Sun SPARCstation). Several programs have been ported from UNIX to the RAP, and we have plans for several more, including a general purpose object-oriented multi-layer perceptron simulation program [Koh] and a Sather language [Omo90] environment. Additionally, the C++ object-oriented interface from UNIX workstations has proven to be an easy way to integrate the RAP into interactive graphics applications and other programs needing a computationally powerful server.

11 Acknowledgements

The RAP Machine project is a group effort by the Realization group at ICSI. The following is a list of Realization group members who contributed to the RAP project.

- Dr. Nelson Morgan: Algorithms and architecture, project management
- James Beck: Hardware architecture, design, and implementation
- Jeff Bilmes: Software architecture design and implementation.
- Phil Kohn: Software architecture design and implementation.
- Dr. Joachim Beer: Preliminary architecture
- Eric Allman: Preliminary software

Chuck Wooters ported the dynamic programming problem for the RAP. And finally, we gratefully acknowledge the support of the International Computer Science Institute.

References

[BBK91] Jeff Bilmes, James Beck, and Phil Kohn. *Installing a RAP System.* International Computer Science Institute, 1991.

[Bec90] James Beck. The Ring Array Processor (RAP): Hardware. Technical Report 90-048, International Computer Science Institute, September 1990.

[Ent] Entropic Research Laboratory, Inc., 600 Pennsylvania Avenue, SE, Suite 202, Washington, DC. *Entropic Signal Processing System.*

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Heu] Heurikon Corperation, 3201 Latham Drive Madison, WI 53517. *Heurikon HK86/V20 Manual.*

[IEE85] IEEE standard for binary floating-point arithmetic. SIGPLAN Notices 22:2, 9-25, 1985.

[KB90] Phil Kohn and Jeff Bilmes. The Ring Aray Processor (RAP): Software Users Manual. Technical Report 90-049, International Computer Science Institute, September 1990.

[Koh] Phil Kohn. *CLONES: A Connectionist Layerd Object-oriented NEtwork Simulator.* International Computer Science Institute. In preparation.

[MB90] Nelson Morgan and Herve Bourlard. Continuous speech recognition using multilayer perceptrons with hidden markov models. In *Proc. International Conference on Acoustics, Speech, and Signal Processing*, pages 413–416, Albuquerque, NJ, 1990.

[MBK+90] Nelson Morgan, James Beck, Phil Kohn, Jeff Bilmes, Eric Allman, and Joachim Beer. The RAP: a Ring Array Processor for Layered Network Calculations. In *Proc. International Conference on Application Specific Array Processors*, pages 296–308, Princeton, NJ, 1990. IEEE Computer Society Press.

[Mor90] Nelson Morgan. The Ring Array Processor (RAP): Algorithms and Architecture. Technical Report 90-047, International Computer Science Institute, September 1990.

[Omo90] Stephen M. Omohundro. *The Sather Language.* International Computer Science Institute, 1990.

[PGTK88] D. Pomerleau, G. Gusciora, D. Touretzky, and H. Kung. Neural Network Simulation at Warp Speed: How we got 17 Million Connections per Second. In *IEEE International Conference on Neural Networks*, San Diego, CA, July 1988.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing. Explorations in the Microstructure of Cognition*, chapter 8. The MIT Press, Cambridge, Massachusetts, 1986.

[Tex88a] Texas Instruments. *Third-Generation TMS320 User's Guide*, 1988. Document Title: SPRU031.

[Tex88b] Texas Instruments. *TMS320C30 Assembly Language Tools*, 1988. Document Title: SPRU035.

[Tex89] Texas Instruments. *TMS320C30 C Compiler Reference Guide*, 1989. Document Title: SPRU034A.

[Win] Wind River Systems, Inc., 1351 Ocean Ave. Emeryville, CA 94608. *VxWorks.*