Unclassified

AD-A193 297

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER 86-02-05
7. AUTHOR(s) Lawrence Snyder and David Socha		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE June 1986
		13. NUMBER OF PAGES 15
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Poker parallel programming environment; Cosmic Cube; retargetting parallel architectures; software portability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a technique for retargetting Poker, the first complete parallel programming enviornment, to new parallel architectures. The specifics are illustrated by describing the retarget of Poker to CalTech's Cosmic Cube. Poker requires only three features from the target architecture: MIMD operation, message passing inter-process communication, and a sequential language (e.g. C) for the processor elements. In return Poker gives the new architecture a complete parallel programming environment which will compile Poker parallel programs, without modification, into efficient object code for the new architecture.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Poker on the Cosmic Cube:
The First Retargetable Parallel Programming
Language and Environment

Lawrence Snyder
David Socha

Department of Computer Science, FR-35
University of Washington

Technical Report 86-02-05

June 1986

Abstract

This paper describes a technique for retargetting Poker, the first complete parallel programming environment, to new parallel architectures. The specifics are illustrated by describing the retarget of Poker to CalTech's Cosmic Cube. Poker requires only three features from the target architecture: MIMD operation, message passing inter-process communication, and a sequential language (e.g. C) for the processor elements. In return Poker gives the new architecture a complete parallel programming environment which will compile Poker parallel programs, without modification, into efficient object code for the new architecture.

Poker on the Cosmic Cube:
The First Retargetable Parallel Programming Language and Environment[†]

Lawrence Snyder
David Socha

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Abstract

This paper describes a technique for retargetting Poker, the first complete parallel programming environment, to new parallel architectures. The specifics are illustrated by describing the retarget of Poker to CalTech's Cosmic Cube. Poker requires only three features from the target architecture: MIMD operation, message passing inter-process communication, and a sequential language (e.g. C) for the processor elements. In return Poker gives the new architecture a complete parallel programming environment which will compile Poker parallel programs, without modification, into efficient object code for the new architecture.

1 Introduction

Software portability for sequential computers became an issue in the early sixties as higher-level languages began to supplant the machine specific assembly languages and as machine varieties proliferated; it was a much harder problem than originally supposed and it remains a serious problem today. By comparison, portability of a parallel program should be substantially more difficult because:

- architectural differences among parallel computers are much more fundamental than among sequential machines, and

- the characteristic that makes portability difficult – the dependence of programs on machine specific features – arises often in parallel computation in order to get good performance.

We say "should be more difficult" because to date there is little experience: There is little production parallel software and there are few truly parallel languages, and few parallel machines. But in spite of the potential problems, there is some reason for optimism.

The Poker[1] language has been retargetted from the Pringle Parallel Computer [2] to the CalTech Cosmic Cube [3]. Thus, programs written for the CHiP [4] family of computers can run on one of the cube family of architectures without modification. This is possible because

- the Poker language uses a reasonably universal program abstraction,

- Poker programs have a (unique) structure that is both visible and simple, and

- the Poker language and environment is structured to make retargetting simple.

There is no impediment to porting the Poker language to other parallel computers as this paper will explain.

The benefit of portable parallel software is obvious: Programs can be written without regard for the underlying architecture. However, portability only guarantees that programs will *function*. With Poker, we are making a stronger claim: Poker programs will run with an efficiency that is comparable to that of programs which were specifically written for that architecture. This leads to a key point about the retargetable Poker parallel programming environment:

> Poker requires a small set of system functions of the host architecture and can thus serve as the definition of the basic software support required of a new parallel computer.

Simply by creating the few basic interface systems that Poker requires, an architecture automatically inherits the available Poker programs and a complete software environment. This vastly reduces the software development efforts for new parallel machines.

2 Overview

Before explaining the details of the retarget, some familiarity with the Poker system must be acquired. Towards this end we present first a high level view of the Poker system pointing out some of the key components and their interactions. Later, we discuss the relevant software and hardware pieces, devoting a section each to: the Poker programming environment, the Cosmic Cube, and the new cross-compiler. Finally, we connect all the pieces and discuss the effects of the new extended system.

Figure 1 shows the relationship between the Poker Parallel Programming Environment and the parallel computers which can be programmed with it. Poker is a sequential system that makes extensive use of a relational database to represent programs. It is written in C[5] and built on top of UNIX[1].

To see how Poker works, focus on two components of the system: the cross-compiler and the debugging environment. The cross-compiler accepts a Poker program as input and produces an object version suitable for execution on one of three machines: the Pringle, the Cosmic Cube, or a simulator/emulator of a parallel machine that runs on whatever sequential machine is hosting Poker itself. The compiled version is then down-loaded to the target parallel computer and executed. During execution, the run-time support of the machine sends tracing information back to Poker's debugging environment so that the programmer can view the execution of the program within the Poker environment.

Thus, one writes and runs programs from an environment that provides flexible interactive graphic support and a view of the program consistent with its definition. During the program debugging activity there is communication between the front-end processor and the back-end parallel machine to facilitate program tracing. When the program is debugged, no tracing is requested and Poker serves as the operating system for the back-end machine, running the program "flat out."

Thus, Poker is both a language and an environment: A sequential system from which to compile, execute, and debug programs on parallel computers. A "port" of the Poker language entails retargetting the cross-compiler and constructing the run-time software to support the communication between the Poker environment and the parallel computer. Only the Poker programs and run-time system get ported to the new machine; the Poker environment runs on a sequential computer.[2] This paper will detail the activities required in crossing the vertical line of Figure 1.

3 The Poker Programming Environment

The Poker programming environment is built around a programming abstraction that is common to non-shared memory parallel algorithms, as described in the next section. However, Poker's interface to the

---

Poker Database

Poker Environment

Programming Environment

Poker program:

graphs   text

Family of Cross-compilers

VAX ...

Debugging Environment

Trace View:

Translation   Translation   Translation

Tracing Information

Emulator/Simulator

VAX

Run-time Support

Pringle Computer

Run-time Support

Cosmic Cube

Run-time Support:

File System Support

Debugger Support

Controller

Base Poker System

Port
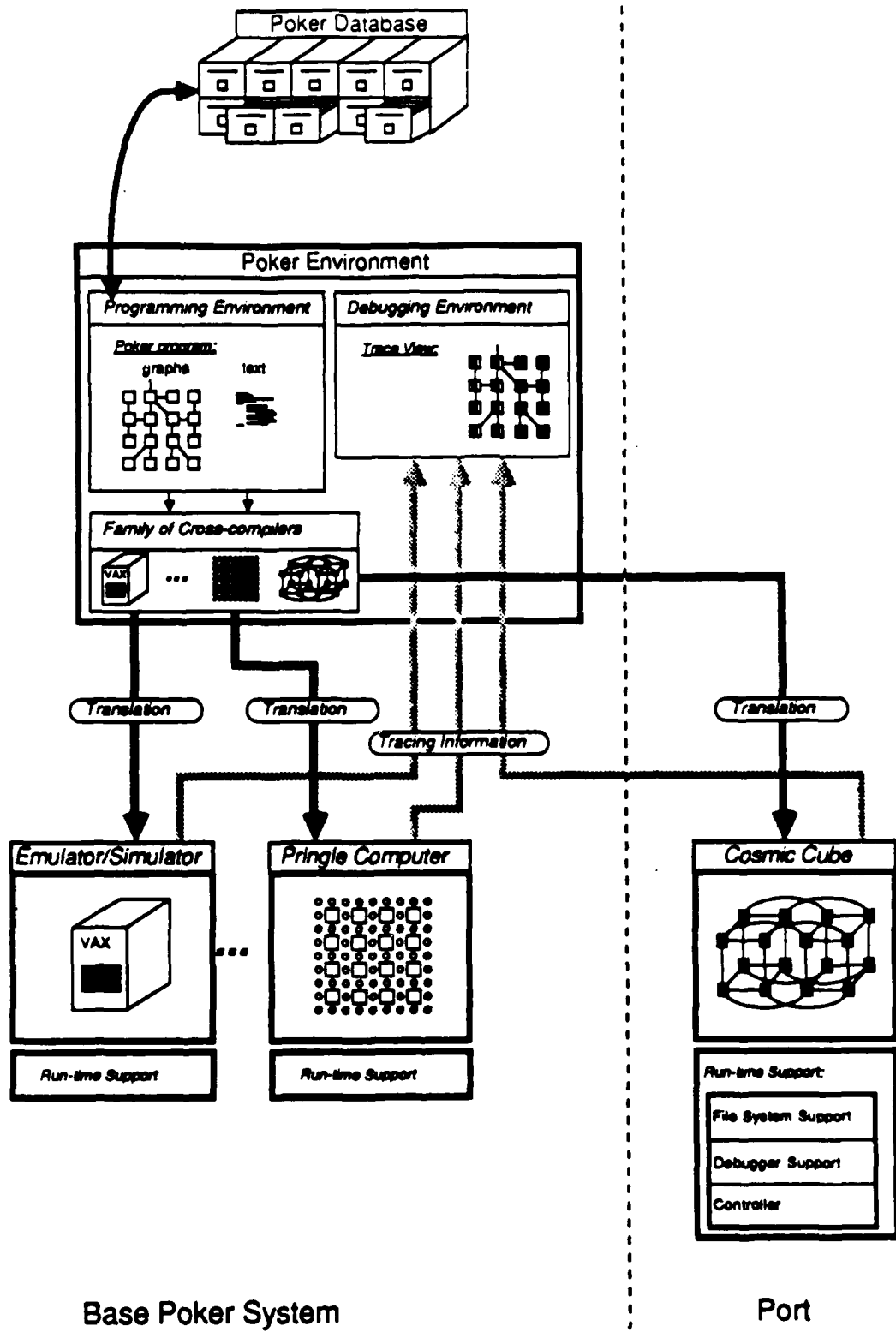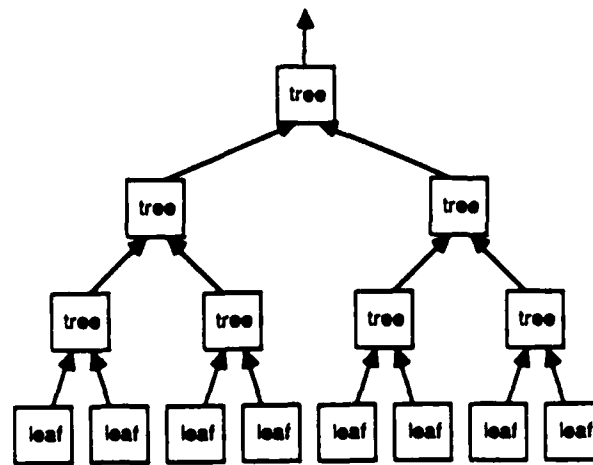
Figure 1: An Overview of the Poker System.

tree: parent <- max (myValue, leftValue, rightValue)
leaf: parent <- myValue

Figure 2: The Maximum Algorithm.

abstraction is quite non-standard. Section 3.2 on Concrete Poker Programs outlines the structure and
semantics of Poker programs. As we shall see, this formulation of parallel algorithms lends itself to easy
retargetting for new parallel architectures.

3.1 Abstract Poker Programs.

A non-shared memory parallel algorithm is conceptualized as a finite graph. The graph's vertices are labeled
with process names corresponding to sequential programs. The graph's edges are of two types: Edges between
vertices are communication paths between processes, and edges "dangling" off of the graph are channels
through which streams of data pass to or from the graph. (Technically, graphs cannot have dangling edges,
of course, but it is convenient to abuse the definition.)

For example, Figure 2 shows the maximum algorithm. The graph is a binary tree. The vertices are
labeled with one of: the name of the leaf process, which passes its local value to its parent, or the name
of the tree process, which receives two values from its children, finds the maximum of these values and its
local value, and passes the result to its parent. The dangling edge is a stream containing a single output
value produced at the root.

A problem is not generally solved by a single algorithm of this form. Rather this form is usually one
"phase" of a computation. The problem is partitioned into a series of phases, each possibly with a different
graph structure. Inter-phase communication is permitted only among those that share the same "location"
as described by a one-to-one mapping function. Values from previous phases may be inherited by the vertex
executing in the corresponding vertex in the next phase.

3.2 Concrete Poker Programs.

Unlike a C or Pascal program, a Poker *phase program* is not a monolithic piece of text. Rather, it is
composed of five components that correspond closely to the abstraction just presented (Figure 3 shows the
five components of the maximum algorithm encoded as a Poker phase program).

- *Communication Graph:* A finite graph with dangling edges. The boxes correspond to processes and
  thus will be the vertices of the graph. The circles, which are switches for the CHiP computer [4], can
  be ignored for the present discussion.

4

Communication Graph
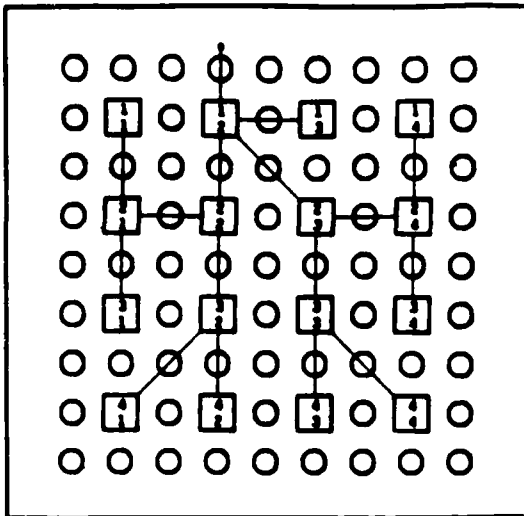
Process Definition

```
code tree;
trace tree().myValue;
ports parent, left, right;

tree()
{
    int myValue, value;

    value <- left;
    if (value > myValue)
        myValue = value;

    value <- right;
    if (value > myValue)
        myValue = value;

    parent <- myValue;
}
```
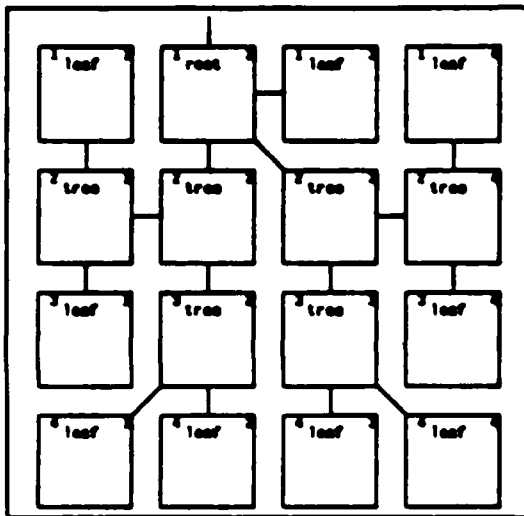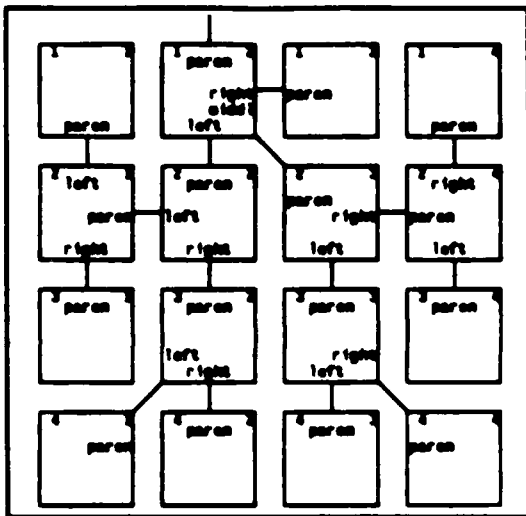
Process Assignment
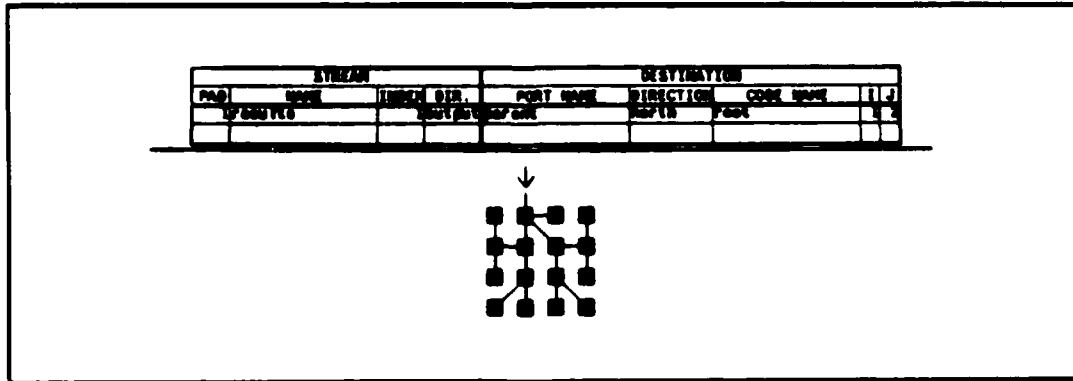
Port Name Assignment

Stream Name Assignment

Figure 3: The five parts of a Poker program.

- *Process Definition:* A (usually small) set of processes written in a sequential language, in this case a slightly modified version of C[5]. These can be thought of as standard procedures with formal parameters called at the beginning of a phase.

- *Process Assignment:* A labeling of the vertices of the graph with the names of the processes and actual parameters, if any, to be executed at that processing site.

- *Port Name Assignment:* A labeling of the edges of the graph, but from the point of view of the vertex, *i.e.* each edge has two names, one for each end, or "port".

- *Stream Name Assignment:* A labeling of the dangling edges of the graph giving names to the input and output streams; these names will subsequently be bound to files.

The correspondence of the five Poker program components to the phase abstraction given above should be clear.

A Poker program is composed of a finite set of phase programs together with an execution scheduler that describes the sequence in which they are to be invoked. Those vertices of different phases which occupy the same location in the Switch Setting View may communicate across phases through the use of inter-phase variables.

Inter-phase variables live for the duration of the Poker program and exist separately from the variables declared local to a phase. The local process codes may access the values of inter-phase variables by the import and export statements:

$$\text{import } local \text{ from } inter\text{-}phase$$
$$\text{export } local \text{ to } inter\text{-}phase$$

Import copies the value of the inter-phase variable into the local variable. Export copies the value of the local expression into the inter-phase variable. This is the only way to pass information between phases. It provides a well-defined interface to inter-phase communication and lets process codes load from and store to inter-phase memory during the course of computation.

Two other features of Poker C are the trace list and inter-process communication. The optional trace list found in the declaration section of each routine specifies the variables whose values will be traced in the debugging environment, if a traced execution of the program is requested. Trace "variables" are not restricted to the conventional "variables" of Poker C. They may include variables, labels, and other items such as the current procedure and depth of recursion.

Process input and output, respectively, are given by expressions of the form:

$$variable \text{ <- } port$$
$$port \text{ <- } variable$$

The values transmitted are simple scalar values from the language, and arrays and structures not containing pointers. Messages are tagged with their type so that process I/O may be type checked using structural type equivalence.

3.3   The Programming Environment.

The Poker system is the set of facilities that assist the programmer in writing and running Poker programs. Poker uses two displays: One is a bit-mapped display having the general form shown in Figure 4 and used for interactive graphical programming of the parallel aspects of the program. The second terminal is used with a standard editor to write the sequential C process text.

Poker stores programs as a database, displaying the program in one of several views [6]. Figure 4 shows the display for the Switch Setting View; the other views are analogous, with the appropriate Poker program constituent displayed in the lower half of the screen. The seven views are

6
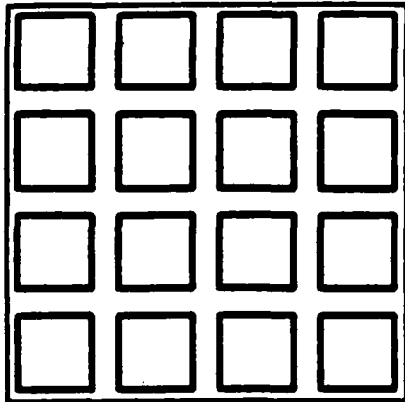
Figure 4: The Poker Display showing the Switch Setting View.

- *Switch Setting View:* Used to define the Communication Graph (Figure 3); the programmer uses a mouse or keypad to draw a picture of the graph by connecting the boxes with lines.
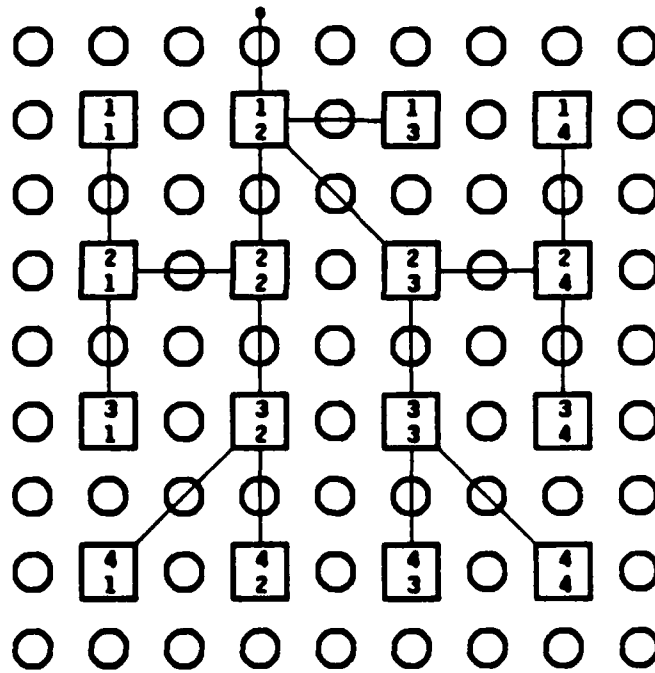
- *Code Names View:* Used to define the Process Assignment (Figure 3); the programmer moves from box to box entering the names of the process and actual parameters, if any.

- *Port Names View:* Used to define the Port Name Assignment (Figure 3); the programmer moves from box to box entering the names of the ports.

- *IO Names View:* Used to define the Stream Names Assignment (Figure 3); the programmer enters the names of the streams and their directions.

- *Command Request View:* Used to compile, assemble, link, load, etc. Poker programs. The bottom of the display shows the progress of the computation.

- *Trace View:* Used to display the execution of the Poker program; the programmer can start, stop, and single step the program while watching the successive changes of the traced variables.

- *CHiP Parameters View:* Used to describe the logical CHiP architecture being programmed. This includes parameters such as the number of processors in the processor grid (16 in Figure 3).

Notice that the display of most of the program constituents includes information defined in other program views; two examples (see Figure 3) are the graph shown in the Code Names View (Process Assignment) which is derived from the graph defined in the Switch Settings View (Communication Graph), and the material in the right half of the table of the IO Names View (Stream Name Assignment) which is gathered from the other views.

4 The Cosmic Cube

The Cosmic Cube [3] provides flexible processing facilities that easily adapt to hosting the more structured Poker abstraction. Each Cosmic Cube program is also a graph: a set of processes, with non-shared memory, communicating through logical links mapped onto the underlying hardware [7]. However, instead of the static graphs of the phase program found in Poker, the Cosmic Cube programs consist of a single, perhaps dynamically changing, graph. Processes may create new communication links (edges), if the creator knows the address[3] of the other process, or destroy old links in order to build the best graph for the moment. This gives the Cosmic Cube programs more flexibility than we need.

This abstraction is implemented on a MIMD non-shared memory computer with a binary n-cube interconnection; the processors and their local memory, or "nodes", sit at the corners of the n-dimensional cube while the edges of the n-cube are formed from physical wires connecting the node processors. Each node may host zero or more processes, living in separate address spaces, communicating by message-passing over the physical wires connecting adjacent nodes. The operating system automatically forwards messages between non-adjacent nodes, preserving the message order between the sending and receiving processes.

A Cosmic Cube program starts as a single process on the host machine[4] connected to one corner of the Cosmic Cube. This host process "spawns" the processes in the initial program graph, placing them on the nodes of the cube and establishing the graph edges by forwarding process addresses to the processes on the cube. These processes may then spawn more processes of their own or create new communication links by sending known process addresses to other processes.

Typically, process codes are written in Cosmic Cube C, a version of the C programming language [5] extended with calls to routines in the Cosmic Cube's operating system, or one of the other extended sequential languages supported for the Cosmic Cube.

---

[3] On the Cosmic Cube a process' address consists of two numbers: the number of the physical node in which it resides, and its process number on that node.

[4] A typical host machine is a Sun Microsystems workstation.

5 The Poker to Cosmic Cube Cross-Compiler

Retargetting the Poker language and porting the run-time system to the Cosmic Cube changed nothing in the existing Poker environment; neither the Poker programs nor the Poker environment needed modification (process codes written in XX [2] are preprocessed into Poker C). Instead we only needed to (1) retarget the cross-compiler to translate Poker C process codes into a single Cosmic Cube C program, using the information in the database to determine the configuration of the resultant graph, and (2) extend the run-time software on the Cosmic Cube to include a few routines interfacing to, and controlling, the Cosmic Cube program. This Cosmic Cube C program from the translator is then treated as any other Cosmic Cube C program, compiling, loading, and executing it using the facilities provided for the Cosmic Cube. The difference is that the extra interface routines know about the Poker system, so that the user of Poker need not know which underlying architecture is executing the user program – no matter which target architecture is used, the creation and execution of the program will be the same. This retarget and run-time system port is the topic of the remainder of this paper.

5.1 Converting Poker C to Cosmic Cube C.

The first task of the cross-compiler is to convert the Poker C process codes into a form acceptable for the processors of the target architecture. There are at least three ways to support a sequential language on the processors of a new architecture. One is to directly generate object code for the target machine's processor elements. A second method is to provide a kernel that runs on the processors of the new architecture and interprets the sequential language (or an intermediary language derived from it). This method could easily support a number of interpretive languages such as LISP [8] and Prolog [9]. The third approach is to translate the source process codes into a language already supported on the new architecture. This last approach works best when the languages are similar in structure, as is the case with Poker C and Cosmic Cube C.

Translating one sequential language into another, source-to-source translation, is well understood. In our case, the only major translation problems come from tracing the variables in the trace list and from reducing all of the Poker phase graphs into one more general and less structured Cosmic Cube graph.

Trace variables are handled by the first part of the cross-compiler which uses a Yacc [10] based C-to-C compiler to insert calls to a run-time trace routine after every instance of a traced variable. The C-to-C compiler does not look for aliases; instead it only inserts a trace command after each assignment whose left-hand-side is an instance of a literal in the trace list. If aliases are a concern, Poker will, at the user's discretion, insert a special trace call after each suspect assignment, exhaustively checking for any changes in the traced variables. Exhaustive traces are expensive at run time but, presumably, a great benefit for debugging.

The C-to-C compiler also replaces instances of the Poker's inter-process communication statements

$$variable \texttt{<-} port$$
$$port \texttt{<-} expression$$

with calls to the Cosmic Cube send() and receive() routines.

The C-to-C compiler is more complex than a simple pre-processor such as UNIX's cpp. To see why, consider tracing a variable that is modified twice in a single expression. We want to trace both changes, but to capture both values of the traced variable we have to insert trace calls *into* the expression. We cannot simply append a trace call onto the expression since the value of the expression may be needed for an assignment or conditional test. Instead, we have to store the expression value in a temporary variable, trace the changed trace variables, and then append the temporary variable to return the expression value.

For instance, given the variables

```
float f;
int i, j;,
```

9

and a request to trace f and j, the C-to-C compiler converts the conditional expression

```
(((f += 4.3) < 10) ?
f++ + j++, i-- :
f--)
```

into

```
(((_tempint = ((f += 4.3) < 10)), _Trace(&f, FLOAT),
        _tempint) ?
f++ + j++, _Trace(&f, FLOAT), _Trace(&j, INT), i-- :
((_tempfloat = f--), _Trace(&f, FLOAT), _tempfloat))
```

where _Trace is a trace routine taking a pointer to a value and a constant telling it the type of the value and _tempint and _tempfloat are reserved variables declared in the enclosing routine.

In the first line, _tempint determines which of the next two expressions to execute: the comma expression before the colon, or the simple expression after the colon. We have to insert a trace of f immediately after the < expression, since if we waited to trace the value of f until after executing the entire conditional expression, we would miss the first value of f.

For the same reason we needed _tempint, the value of the expression was used outside of the expression, we also need to use _tempfloat to save the value of the conditional expression before tracing f's new value.

Clearly, this is not a simple textual substitution. Before we insert a temporary variable, we have to know the type of the expression. This requires keeping a symbol table, to hold the types of the variables, and using a parse tree to calculate the types of expressions. In other words, the C-to-C compiler is truly a compiler even though the languages it consumes and generates are nearly identical.

The end result of the C-to-C compiler is two code segments from each of the Poker C process codes. One segment contains information about the inter-phase variables imported to, and exported from, that process code. This other segment contains the routines defined for that process code. These code segments are used by the cross-compiler, as discussed in the next Section.

5.2 Compiling the Poker Database.

The second part of the cross-compiler combines the pieces of the newly translated code with the rest of the information in the Poker program's database to create a single Cosmic Cube C program. The first problem here is to figure out how to collapse the phases of the Poker program into one graph, while maintaining efficient inter-phase communication[5]. This is easily achieved by the technique suggested in Figure 5. When the phases are stacked as in Figure 5 the processes above one another are exactly those processes that share inter-phase variables. Combining them into one "aggregate" process keeps the Poker processes in a single process space. Inter-phase variables are globally declared for that process so that inter-phase communication requires no extra computation.

Building these aggregate processes is done as follows. The cross-compiler computes the inter-phase data space of each aggregate process from the inter-phase variables required by each of the Poker processes being placed into the aggregate process. The main() routine of each aggregate process is simply an infinite loop containing a call to the controlling host process asking for the number of the phase to execute, followed by a switch statement to call the main routine of the appropriate phase (see Figure 6). Since Poker C does not allow externally declared variables, the only potential source of name conflicts among different phases come from the routines, typedef's, and externally defined structures and unions. Prefixing these names with a unique phase ID eliminates any possible name conflicts. Linking the main() code with the routines from the phases and routines supporting the calls to receive(), send(), and the like, results in the Cosmic Cube C code for the aggregate process.

The last problem with the aggregate processes is mapping the edges of the different phases onto one graph. Since the Cosmic Cube does not have any restriction on the degree of the vertices (processes) in its

---

[5] Keeping efficient inter-process communication is a much more complex issue, discussed in the Results.
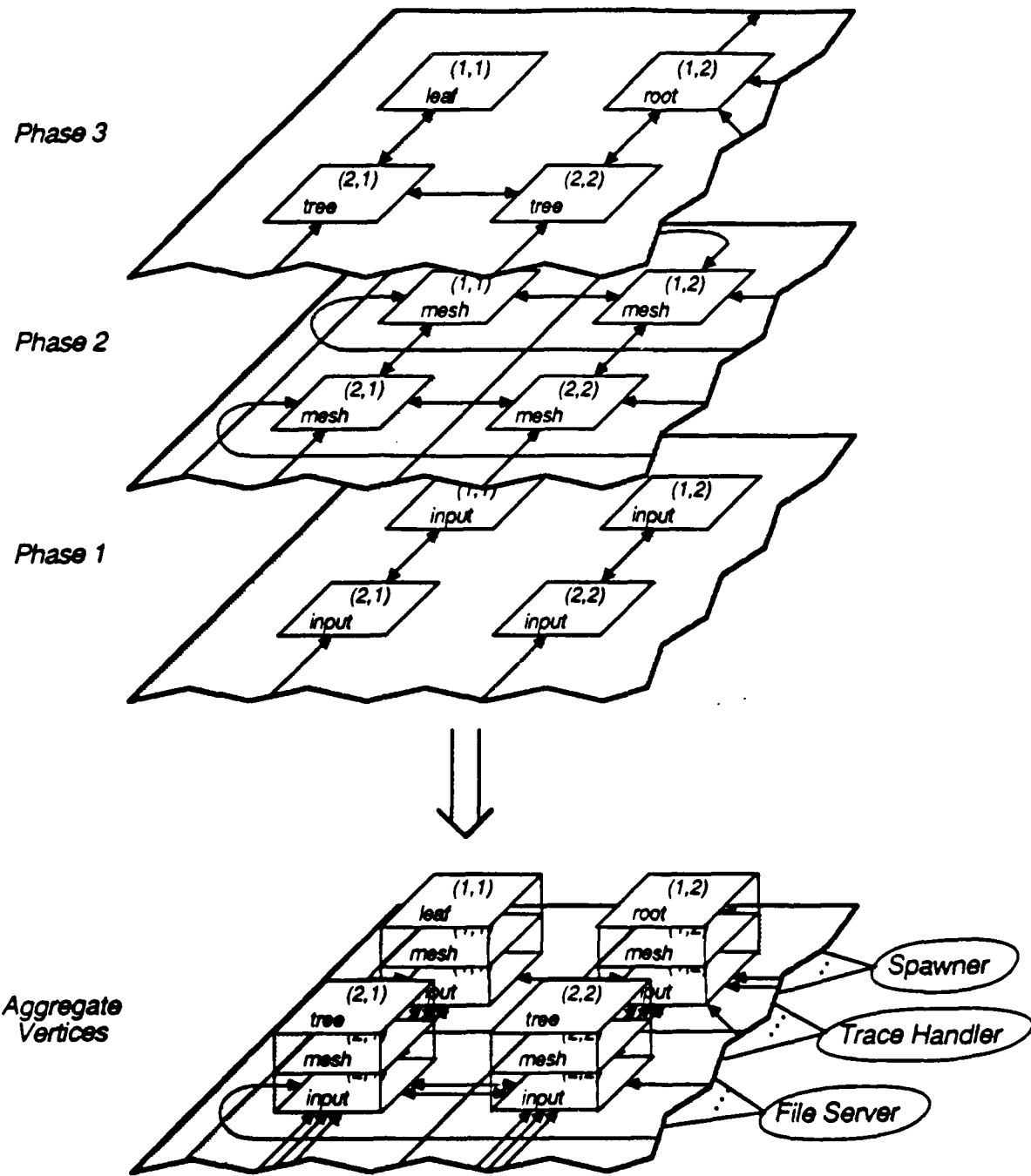
Figure 5: Collapsing the phases of a Poker program to create the aggregate processes.

Vertex Codes:

```
code phase1;
trace phase1().i, phase1().foo;
ports in, out;

main()
{
    int i, foo;

    import foo from foobar;
    sub1(foo);
    /* rest of main(), phase 1 code */
}

sub1(val)
    int val;
{
    int temp;

    /* computation... */
    export temp to result;
}
```

code for phase 1:

```
code phase2;

main()
{
    /* code for main(), phase 2 */
}

sub1()
{
    /* code for sub1(), phase 2 */
}
```

code for phase 2:

Aggregate Vertex Code

```
/* phase-global variables */
int _global_foobar, _global_result;

main()
{
    while (TRUE)
    {
        receive (SPAWNER, nextPhase);
        switch (nextPhase)
        {
            case 1:  /* phase 1 */
                phase1();
                break;
            case 2:  /* phase 2 */
                phase2();
                break;
            case EXIT:   /* terminate */
                _ExitFromProgram();
        }
    }
}
```

```
phase1()
{
    int i, foo;

    foo = _global_foobar;       /* import */
    _Trace(&foo, INT);
    sub1(foo);
    /* rest of main(), phase 1 code */
}

_p1_sub1(val)
    int val;
{
    int temp;

    /* computation... */
    _global_result = temp;       /* export */
}
```

Routines for phase 1

```
phase2()
{
    /* code for main(), phase 2 */
}

_p2_sub1()
{
    /* code for sub1(), phase 2 */
}
```
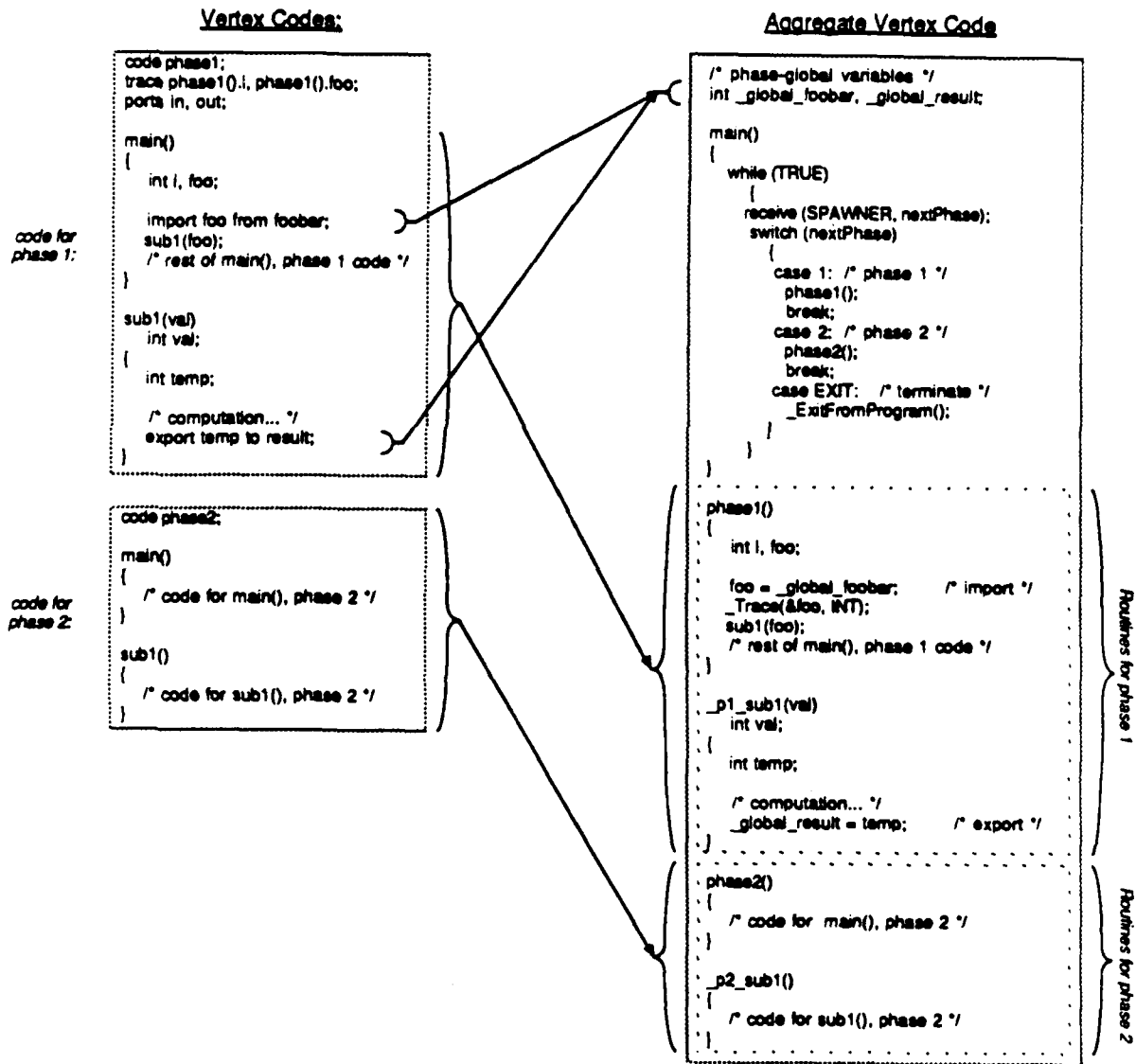
Routines for phase 2

Figure 6: Packaging the Poker C process codes into a Cosmic Cube C aggregate process.

program graph, we can project all of the phase graphs onto one graph. In actuality, each aggregate process gets a different logical interconnection map for each phase. The map holds the eight pairs of (process address, port number) that correspond to the terminal ends of the wires connected to the ports of the process for that phase. Since process addresses are known only after the processes are placed on the Cosmic Cube nodes, these tables are initialized at run time.

This packing results in extremely efficient inter-phase communication and phase change. One of the advantages of Poker programs is that the phase graphs are known at compile time so that the processors do not have to expend any run-time effort constructing or modifying the program graph or dynamically allocating more processes.

After spitting out the aggregate vertex codes, the cross-compiler runs them through the Cosmic Cube C compiler to get executable versions ready to be placed on the Cosmic Cube.

5.3 Extending the Cosmic Cube's Run-time System.

The third task of the cross-compiler is to augment the operating system of the target computer with special processes implementing the features we need for the source language. In the case of Poker, we have three such special processes: the File Server providing file system support, the Spawner acting as the controller, and the Trace Handler providing debugger support. All three of these special processes live on the host computer attached to the Cosmic Cube.

- *Spawner:* The Spawner has two tasks:

    - *Initializing a Poker program onto the Cosmic Cube.* This includes:
        * *Mapping aggregate processes to Cosmic Cube nodes.* Currently, the aggregate processes are mapped naively onto Cosmic Cube nodes, with no attention to the interconnection between the processes. Better allocations are often possible, but the realization of one of these is a complex matter discussed in the Results.
        * *Placing the aggregate processes on the Cosmic Cube nodes.* The Spawner, takes the executable codes for the aggregate processes and places ("spawns") them on the Cosmic Cube nodes.
        * *Initializing the edge maps of the "aggregate" processes.* After the Spawner places all of the aggregate processes on the Cosmic Cube the Spawner sends each aggregate process a message containing the addresses of the processes on the other end of its edges, so that the aggregate processes can initialize their edge mapping tables. The Cosmic Cube operating system provides these addresses as a result of the Spawner placing the aggregate processes.

    - *Controlling execution order of the phases.* The Spawner is the Master Control Program that sends messages to all of the aggregate processes telling them which phase to execute. When each aggregate process finishes its phase code, it sends a "Done!" message to the Spawner. The Spawner waits to hear the "Done!" messages from all of the processes before sending the next "Execute phase n!" message to all of the aggregate processes.

- *File Server:* Processes access Poker's file system through dangling edges. The File Server keeps "input" edges filled with values from files and stores values from "output" edges into files. All messages on dangling edges route through this File Server.[6]

- *Trace Handler:* Trace variables are variables whose values are always known to the outside world where a programmer may view them in a special Trace View within Poker. The Trace Handler collects messages from the processes indicating new values for the traced values and sends them to Poker's debugging environment. These values come from send() statements that the C-to-C compiler inserts after each assignment to a traced variable.

---

[6]This is, of course, a potential bottleneck. Multiple File Servers could be used, but the architecture of the Cosmic Cube still requires all messages between the Cosmic Cube and the host machine, and thus the file system, to pass through the single physical edge connecting the host computer to the Cosmic Cube. Without additional hardware or a different basic IO design, Cosmic Cube programs risk becoming IO bound even with "reasonable" IO overhead.

5.4 The Cosmic Cube Poker Program.

In summary, the Cosmic Cube program corresponding to a Poker program consists of the aggregate processes on the Cosmic Cube nodes and three special processes running on the host computer: the Spawner, the File Server, and the Trace Handler. These processes implement the run-time support, while Poker provides the programming environment, traces program execution via the Trace Handler, and cross-compiles the Poker program to create the Spawner, the File Server, and the aggregate processes that run on the Cosmic Cube.

6 Results

6.1 Machine dependencies.

Most of the Poker system is machine independent. The machine dependent pieces fall into three parts:

- *The Cross-compiler.* The C-to-C compiler inserts calls to run-time routines for inter-process communication, inter-phase communication, tracing, and execution control. Changing these calls requires slight changes to the C-to-C compiler. If the target processors support C, the C-to-C compiler should not need any changes. However, if the target language is not C, the C-to-C compiler may need major reworking.

- *The Run-time system.* The routines supporting the interface between the Poker environment and operating system of the target processors are highly machine dependent. Large parts of these may have to be written from scratch.

- *The mapper from logical to physical interconnections.* Not only is this the least efficiently implemented feature of the system, as described below, but it also is extremely machine dependent.

Changing these three parts is sufficient to retarget the Poker language to a new architecture.

6.2 Mapping interconnections.

When the programmer defines a communication graph in Poker and it is run on the Pringle (or, one day, the CHiP Computer), the graph is "directly implemented" in the sense that the figures produced in the Switch Setting View will be quite literally compiled into the object code of the machine. This is because the machines are configurable: The switches route messages according to the interconnection drawn in the Switch Set View. For any fixed-interconnection machine, such as the Cosmic Cube, the situation is different: The communication graph, which will be considered the logical communication structure, must be mapped onto the physical interconnection structure of the architecture. Of course, programmers *implicitly* perform this mapping when programming in other parallel languages.

As described in Section 5.3 on extending the Cosmic Cube's run-time system, the mapping is done by the Spawner using an arbitrary allocation. If the programmer defined a logical cube graph, the system might not, as things now stand, allocate it so that logically adjacent vertices are physically adjacent. In general, the best allocation is not so obvious, yet the system should try to minimize the distance between communicating processes. The problem of automatically finding an optimal allocation instead of our *ad hoc* allocation remains unsolved. Moreover achieving optimality is complicated by multiple phases with the one-to-one correspondence of vertices between phases; a good mapping for one phase may be quite poor for another phase of the same algorithm. We are hopeful that the work of Berman and colleagues [11] will lead to an automated solution, though the retargetting in no way depends on Berman's software; it only improves the quality of the end result. Presently, we advocate a programmer assisted mapping, but we have not yet provided the software for it. No matter what enhancement is chosen, there is an entry point in Poker for the appropriate software.

7 Conclusion

We have shown that it is easy to port Poker to a new parallel architecture, the Cosmic Cube, in such a way as to produce object code as efficient as any written for that architecture. Furthermore, Poker can easily be ported to *any* parallel computer simply by providing three basic features for the new architecture:

- MIMD operation,

- message passing facilities, and

- a compiler for the process codes.

These are very modest requirements considering that a small amount of work would then provide the new architecture with a complete, easy to use programming, debugging, and cross-compiling environment. as well as all of the existing Poker parallel programs. It is for this reason that we claim that Poker can be the definition of the basic software support required by a new parallel computer.

References

[1] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[2] Alejandro Kapauan, Ko-Yang Wang, Dennis Gannon, Janice Cuny, and Lawrence Snyder. The Pringle: an experimental system for parallel algorithm and software testing. *Proceedings of the International Conference on Parallel Processing, IEEE*, 1–6, 1984.

[3] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[4] Lawrence Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[5] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Academic Press, New York, 1978.

[6] Lawrence Snyder. *Poker (4.0) Programmer's Reference Guide*. Technical Report 86–05–04, Computer Science Department, University of Washington, May 1986.

[7] Wen-King Su, Reese Faucette, and Chuck Seitz. *C Programmer's Guide to the Cosmic Cube*. Technical Report 5203:TR:85, Computer Science Department, California Institute of Technology, September 1985.

[8] P. H. Winston and B. K. P. Horn. *Lisp*. Addison-Wesley, 1984 (second edition).

[9] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[10] Stephen C. Johnson. *Yacc – Yet Another Compiler Compiler*. Technical Report Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[11] Francine Berman, Michael Goodrich, Charles Koelbel, III W. J. Robison, and Karen Showell. Prep-P: a mapping preprocessor for CHiP architectures. *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

END

DATE

FILMED

DTIC

JULY 88