

Communication in a Tree Machine

Sally A. Browning

Bell Laboratories
Murray Hill, New Jersey 07974

Charles L. Seitz

Computer Science Department
California Institute of Technology
Pasadena, California 91125

ABSTRACT

Communication assumes a progressively dominant and limiting role in VLSI because it becomes relatively more expensive in chip area, signal energy, and time. The principle of locality becomes all important to integrated systems design, and implies that larger single processors are not the route to performance improvements. One computer architecture that can exploit the capabilities of VLSI is an ensemble of small processors operating concurrently.

The tree machine is such a structure. Each of the many processors in the binary tree can communicate directly only with its parent and two children. However, the tree is programmed as if each processor had an arbitrary number of descendants, and the programs are compiled into code for a binary tree. We describe the communication structure of tree machine programs, the compilation process, and the underlying hardware.

Jan 30, 1981

Communication in a Tree Machine

Sally A. Browning

Bell Laboratories
Murray Hill, New Jersey 07974

Charles L. Seitz

Computer Science Department
California Institute of Technology
Pasadena, California 91125

1. The Tree Machine Architecture and Project

As Very Large Scale Integration (VLSI) becomes a reality, we have the opportunity and motivation to build entirely new kinds of computers. The traditional view of a single large processing unit physically separated from a single large store by a memory bus is certainly realizable in VLSI. But, as Sutherland and Mead⁵ have observed, this architecture requires too much global communication to be a good fit to VLSI. It is also a carryover from the era in which processing and storage were best implemented in different technologies.

Most of the space on an integrated circuit is occupied by the wires that carry control and data to the functional blocks. It is no longer sufficient to spend most of the design effort on the individual cells and leave the wiring until the end. Rather, the interconnection must be considered an integral part of the design. As Seitz⁴ has pointed out, making larger scale single processors in VLSI scaled to submicron dimensions becomes self-defeating: the diffusion delay in a wire scales up quadratically while the delay of a MOS switch scales down linearly. Thus, the ratio of communication to switching delay scales up as the third power of the scaling factor, and rapidly becomes a dominant design factor for metal sizes below about one micron. Faster signal paths on higher, thicker layers of wider metal runs are possible and a likely feature of new technologies, but will be a limited resource and will require larger drivers built on the diffused, polysilicon, and thin metal layers.

Together, these observations might be called the principle of locality for VLSI. If a signal must travel between two widely spaced points, it will either go slowly, or require extra area for bigger drivers and thicker wires. The incentive to design chips with local communication is strong.

The tree machine architecture¹ provides a general purpose computing environment while capitalizing on the properties of VLSI. The tree machine is a collection of very small processors connected together as a binary tree. The processors are identical, and each has its own memory. There is no global communication, only communication between parent and child in the tree, and between the root of the tree and the external world. This architecture gives rise to integrated circuits that have regular interconnect, local communication, and many repetitions of a single processor design. These integrated circuits, in turn, can be assembled in regular patterns at the printed-circuit board and backplane level to construct machines with thousands to hundreds of thousands of processors.

The tree machine is general purpose because it is programmable. A varied collection of algorithms have been designed that take advantage of the available concurrency. In each case, the time complexity of the algorithm is roughly equal to the number of data elements in the problem. Thus, sorting can be done in linear time, and matrix operations in $O(n^2)$ time. Many graph problems, including some that are NP-complete, can be solved in $O(\text{edges})$ time, provided the tree machine has enough processors. Problems that require an exponential growth in resources, like the NP-complete problems and divide-and-conquer algorithms, are a good fit for the tree machine architecture: each additional level in the tree doubles the number of available processors.

This paper describes work done during the summer, 1980, at Caltech. The first design has one processor per chip; As processing with finer design geometries becomes available several processors and their memories will be put on a single chip. The goal is to have a tree machine built and tested by the fall of 1981, and a machine of at least 1023 processors working in 1982.

A feature of the tree machine that makes it particularly attractive for implementation in a research environment is that it can be viewed as a recursive structure. Each node is the root of a subtree. Thus, once the root of the tree has been tested, it can test, recursively, the rest of the tree. Once a bootstrap program has been loaded into the root, it can load the other processors. The importance of simple testing and loading procedures increases with the number of transistors on an integrated circuit.

2. Programming a Tree Machine

Tree machines are programmed in a high level language resembling Hoare's "Communicating Sequential Processes" (CSP) notation.³ The notation, TMPL, is described in full elsewhere;¹ here we will look only at the communication primitives. This paper describes the tree machine communication protocol, and the mapping between the programmer's view of communication and the hardware. Thus, it is in large part a discussion of the TMPL compiler, whose major parts are shown in Figure 1. TMPL programs are written for trees with arbitrary fanout, and transformed by the compiler into source code for a binary tree; this is the MAP step, and is described in section 3.

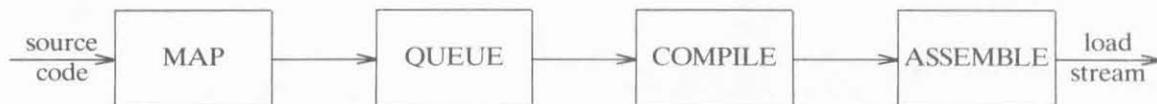


Figure 1. TMPL Compilation Process

While TMPL supports four communication primitives, the hardware implements only two of them. We remove the other two in the QUEUE step. This mapping is discussed following a presentation of the hardware implementation. The final two compilation steps are familiar: the source code is translated into machine code in the COMPILE step, and a load stream is composed in the ASSEMBLE step. These two processes are not described. We begin with a brief presentation of the notation, concentrating on the constructs used in communication.

2.1. Communication Primitives in TMPL

The basic building block of TMPL is the processor definition. Each definition describes a self-contained computational unit that communicates through ports with other processors. A processor definition includes the naming of one *external port* to its parent in the tree, and an arbitrary number of *internal ports* to descendents. Communication statements mention port names through which the message will pass, rather than naming target processors. As a result, processor definitions are written without regard to the eventual connection plan of the tree. A processor expects to follow a specific communication protocol when accessing a port. Any processor that follows the same protocol can be connected to the other end of the port.

This definitional locality makes possible a parts kit of standard processor definitions. Each part is a processor, or tree of processors, that can be completely characterized by the behavior at its ports. As long as the expected messages are sent and received, the parts will work anywhere in the system.

Inter-process communication can be specified in two ways, either as an imperative statement, or as a conditional expression. The statement form results in the processor being blocked until the communication is successfully completed. The conditional form appears as part of a loop or case statement, and is performed only if both processors communicating along the specified port are ready to exchange the message.

Syntactically, message statements and expressions are identical; the general form is shown below. We retain Hoare's notation for the direction of the communication: ? indicates input, and !

is used for output.

$$\left[\begin{array}{c} \text{port} \\ \text{list of ports} \end{array} \right] \left[\begin{array}{c} ? \\ ! \end{array} \right] \text{message}(\text{arguments})$$

They are distinguished by context: message statements can occur wherever a statement is valid, while message expressions are legal only in guards, described below.

A communication involves two processors connected via a port. An output request to the port from one processor must match up with an input request for the same message from the other processor in order for the communication to take place. Either the output or the input can be done conditionally, but not both. This restriction prevents kind of deadlocking: the "after you, after you" situation that arises when neither processor will commit itself to the conditional exchange.

TMPL provides compile-time checking for occurrences of the illegal conditional-to-conditional communication by requiring that message names be typed. These types specify how the message will be used: **imp** for imperative mode, and **cond** for conditional. The message names, types, and directions (input or output) are made externally available. When the tree connection plan is read and the processors linked through ports, the message interfaces are compared. Illegal communications, as well as messages that cannot be paired up, are flagged.

For example, suppose we have three processor definitions called A, B, and C. Message ports, names, types, and directions used in the three processors are shown below, in Figure 2.

Processor	Port	Port Type	Name	Type	Direction	Arguments
A	parent	external	load	cond	?	1
	parent	external	unload	cond	!	1
	left	internal	load	imp	!	1
	left	internal	unload	imp	?	1
	right	internal	load	imp	!	1
	right	internal	unload	imp	?	1
B	parent	external	load	cond	?	1
	parent	external	unload	imp	!	1
C	parent	external	load	cond	?	1
	parent	external	unload	cond	!	1
	left	internal	load	cond	!	1
	left	internal	unload	imp	!	1

Figure 2. Message Descriptors

If we connect the left port of A to B's port, we find that A will send *load* messages, and B will receive them; similarly, B will send *unload* messages and A will receive them. In addition, all communications between the two are either imperative to imperative or imperative to conditional, and thus valid. A connection between the left port of C and B produces invalid communications. While C outputs *load* messages and B receives them, both are doing it conditionally. And the *unload* messages have the same direction: both processors want to send the message. Thus, B and C cannot be connected using the left port of C. Incidentally, the protocols match up if B and C are connected through their parent ports, but in order to preserve the tree structure, connections must always be made from an internal port to an external one, that is, from parent to child.

2.2. Message Statements and Expressions

Message statements are a familiar concept, resembling subroutine calls: the processor executing the statement is blocked from further execution until the communication is successfully completed. Message expressions, however, are more complicated. The following paragraphs describe them in more detail, beginning with a description of the statements in which the message expressions may

appear.

As in Dijkstra's notation,² TMPL has generalized loop and conditional statements made up of a set of guarded commands. A guard is an expression; the command will be executed only if the guard has the value **true**. A TMPL guard can be a logical expression, a message expression, or a combination of the two. A single guard can contain at most one message expression. Each of the following lines contains a valid guarded command.

$$i < 9 \text{ AND } i \geq 0 \rightarrow j := j * 10 + i$$

$$\text{NOT found AND } p?arc(i,j) \rightarrow \text{found} := \text{start} = i \text{ AND } \text{end} = j$$

$$\text{left, right?answer} \rightarrow p!answer ; \text{count} := \text{count} + 1$$

The syntax for loop and conditional statements differs only in the braces used to enclose the set of guards: curly ones are used for loops, and square ones for conditionals. Semantically, however, the two statements are quite different. A loop statement is executed repetitively until all guards are false. A conditional statement is executed exactly once: at least one guard must be **true**, otherwise the statement and the program will abort. In both statements, if more than one guard is **true**, a nondeterministic choice will be made. For example, if $i=5$ when the conditional statement below is entered, all three of the guards are true. We cannot assume that the first one will be chosen, but must settle for a random choice.

$$[\begin{array}{l} i \leq 9 \rightarrow \text{SKIP} \\ | i > 0 \rightarrow i := -9 \\ | i = 5 \rightarrow p!five ; \text{found} := \text{found} + 1 \end{array}]$$

The nondeterministic property of the loop and conditional statements is an integral part of programming a tree machine. The processors act independently; a pair of them is synchronized by communication. Because a processor has three ports to other processors and knows nothing of the timing characteristics of its neighbors, a TMPL program can describe only the *sequence* it will use to access the ports. In many cases, a message could arrive on any one of a set of ports, giving rise to a set of guarded commands, each one triggered by communication on a specific port. For example, the conditional statement below has message expressions for all three ports. We can make no statements about the order in which the guards become true: it depends both on when the ports become active, and on the choice between **true** alternatives.

$$[\begin{array}{l} \text{bus, left, right?done} \rightarrow \text{active} := \text{active} - 1 \\ | \text{left, right?notDone} \rightarrow \text{active} := \text{active} + 1 \end{array}]$$

Conditional and loop statement guards are the only place that message expressions may appear. They may be combined with the familiar logical expressions to make more complicated expressions. While an individual guard can contain at most one message expression, a guard set can contain many, and can mix message expression guards with those containing only logical expressions. If logical and message expressions are combined in a guard, the logical expression must appear first, and is evaluated first. Because the message expression can have the side effect of doing an input or output as it is evaluated, it is examined *only* if the value of the logical expression is **true**.

Like logical expressions, message expressions are evaluated. Unlike logical expressions, they can assume one of *three* values. **False** signifies that the communication cannot be done, and is assigned whenever some *other* communication is pending on the selected port. A message

expression is **true** if the communication is possible: the port is waiting for exactly this message. The **maybe** value is assigned to the expression if it is neither true nor false, that is, the port mentioned in the message expression has no pending communications.

The key to evaluating a message expression is the restriction that at least one of the two processors involved in a communication must use the imperative form. The message *statement* forces the communication, making the issuing processor and associated port busy until the transfer has been completed successfully. Thus, a processor that is evaluating a set of message *expressions* has only solid, imperative communications to match up with. The state on the busy ports won't change while a decision is being made.

3. Mapping Arbitrary Fanouts Onto a Binary Tree

TMPL programs are written as if the processor had as many immediate descendents as needed. In fact, the underlying architecture is a *binary* tree, and arbitrary fanouts are simulated. A processor definition that declares more than two internal ports to children becomes the root of a *composite* processor, shown in Figure 3. Several layers of the tree are used to provide the required number of descendents, and the intermediate processors, called *padding* processors, are provided with a skeletal program that allows them to pass messages between the parent and children. The process of generating those skeletal programs is the subject of this section. We begin with some guidelines for the mapping algorithm, look at a pair of *routing numbers* that can uniquely identify a descendent processor, and finish with a case by case treatment of the seven kinds of communication that the padding processors must handle.

3.1. The Mapping Constraints

The constraints we place on the mapping algorithm have distinctly different flavors. The first one reflects the underlying system architecture, while the second constraint is an arbitrary design decision.

First, there is no wildcard message name that matches everything. Because message expressions must eventually be evaluated as **true** or **false**, the hardware knows about message names. Thus, each padding processor must be tailor-made for the problem and will handle only those messages that can pass between parent and child. If the parent and children exchange *load*, *arc*, and *answer* messages, the padding processor must have message statements or expressions for *load*, *arc* and *answer* messages as well.

Our second constraint is an attempt to minimize the amount of information the mapping algorithm needs: we allow the program for the parent processor to be modified in the mapping process, but not that of the child. It is clear that the code in the composite processor will require modification: it specifies communications on more ports than it really has. But it is not necessary to know at compile time what is connected to the ports.

Several benefits arise from this constraint. We can compile a set of padding processors based only on information available in a single processor definition. The resulting composite processor has precisely the same communication characteristics as the unmapped processor. Thus, assertions about the communication properties of the original processor hold for the composite.

3.2. The Routing Numbers

We use two integers, a *depth finder* and a *path*, to uniquely address a specific descendent of a composite processor. The depth finder indicates whether or not the message has reached a "leaf" of the composite. The path selects the left or right branch at each processor in the composite. New values for the depth finder and path are calculated at each level in the composite processor.

The depth finder measures the number of padding processors that must be traversed to reach the bottom of the composite processor. Thus, the depth finder is not a measure of distance from the root; rather, it is the distance from the leaves. Suppose we are simulating a fanout of n . The composite processor will contain $n-1$ processors: the root of the composite supplies two connections

to children, and each additional padding processor adds two connections while occupying an existing one, for a net gain of one connection. Thus, a composite processor with fanout n contains one copy of the modified parent processor and $n-2$ padding processors. As we descend in the composite processor this count can be recalculated to describe the number of padding processors in the composite if the current one is the root. At the leaves of the composite, the depth is zero.

The rule for calculating the depth finder value depends on the fact that unbalanced composites are heavy on the left side, as shown in Figure 3. The algorithm for calculating the depth finder is given below. It is applied recursively for all subtrees in the composite processor: each descendent is the root of a subtree, and supplies the value for $depth_{parent}$ to the next level.

$$\begin{aligned}
 depth_{parent} &= n-2 \\
 depth_{left} &= \frac{depth_{parent}-1}{2} \\
 depth_{right} &= \frac{depth_{parent}-2}{2}
 \end{aligned}$$

The second argument specifies a path to the desired child, and relies on the allocation of logical to physical ports shown in Figure 3. The numbering of logical ports starts with 1, with odd ports assigned to the left side, and even ones to the right. This rule is applied recursively until all leaf processors have binary logical and physical fanout.

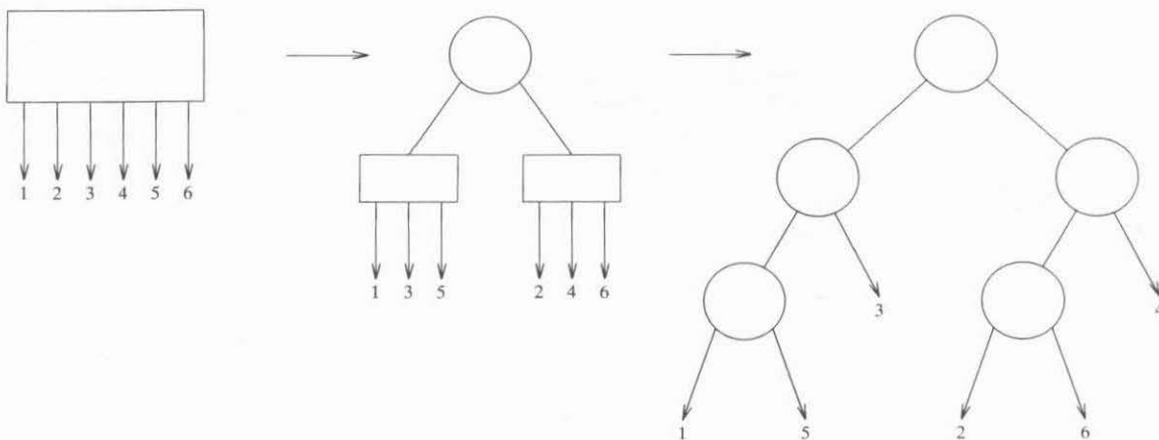


Figure 3. Mapping Arbitrary Fanouts onto a Binary Tree

Suppose the processor has a logical fanout of n , and wants to communicate with its i^{th} child. The path is initially set to i . At each node in the composite, odd path values are passed to the left, and even values to the right. A new path value is computed at each level according to the algorithm below. This, in conjunction with the depth finder, can be used to uniquely select a descendent.

$$\begin{aligned}
 &path_{parent} := i \\
 &[\text{odd}(path_{parent}) \rightarrow path_{left} := \frac{path_{parent} + 1}{2} \\
 &| \text{even}(path) \rightarrow path_{right} := \frac{path_{parent}}{2} \\
 &]
 \end{aligned}$$

Two routing numbers are needed to locate a specific processor because of the asymmetry of the composite processor. Paths from parent to child are not always the same length. In Figure 3, for example, it takes three steps to reach the sixth child, and only two to find the third one. Both the identity and depth of a descendent are essential information, and cannot be encoded in a single number.

3.3. Generating Programs for the Padding Processors

Padding processor programs are generated based on the parent's view of communication. We do not have access to the programs for the descendent processors, and must retain the original message interface in the fully padded composite. A list of message statements and expressions that the parent program might utilize follows:

1. imperative output
2. imperative broadcast output
3. imperative input
4. imperative broadcast input
5. conditional output
6. conditional input
7. conditional broadcast input

Note the absence of conditional broadcast output, better described as the case where all descendents from a node are ready to input the same message. This state can be expanded into a guard that is the logical AND of a set of messages expressions, one for each descendent. Since guards cannot contain more than one message expression, we do not implement conditional broadcast output.

Each of the valid cases is discussed briefly below. The general strategy is this: the parent processor definition is modified to communicate with two descendents through internal ports l and r . Two, one, or none of the routing numbers are appended to the message, as needed. There may be extra messages used for synchronization between parent and child: the message no longer travels directly.

Each padding processor is given the program segment required to read or write the messages expected by the parent. If any routing numbers have been added to a message, they must be stripped off before being handed to the child processor: it is expecting the original message. The depth finder, which records the number of padding processors yet to traverse, is zero at the bottom of the composite processor.

Imperative output directed at a specific processor requires both routing numbers to locate the receiver. Imperative broadcast output communication requires no additional arguments or messages. The message is allowed to spread throughout the composite processor, since it is directed at all descendents.

Imperative input from a specific processor requires both routing numbers, like the corresponding output command. In fact, any time a specific processor is mentioned in a communication statement, both numbers are used to locate it. Imperative broadcast input asks for a message from any one of its children. Note that the statement $c(*)?msg$ is equivalent to a conditional statement:

```
[ c(1)?msg → SKIP
  | c(2)?msg → SKIP
  . . .
  | c(n)?msg → SKIP
]
```

Remember that guards are not evaluated in any particular order: a non-deterministic choice is made among those that evaluate to **true**.

The imperative broadcast input is treated as if it were a conditional statement: each child is

asked whether it has something to give the parent. If not, the next child is interrogated until one is found that is trying to send the matching message. The fact that the input is imperative guarantees that there will be at least one child that wants to send the message.

The template for imperative broadcast input relies heavily on the fact that message expressions are evaluated. The pair of statements that follow demonstrate a technique for finding out how a message expression was evaluated.

```
got := FALSE ; { NOT got AND !?msg → got := TRUE }
```

The loop will execute zero or one times, and the boolean value *got* can be used to find out what happened. The loop will be stuck evaluating the guard until some communication is initiated on the port called *l*. If the communication is a request to output *msg*, the value of the guard is TRUE, and *got* becomes true. The loop will not be executed again because of the *NOT got* phrase. If, on the other hand, the communication initiated on port *l* does not match, the value of the guard is false, and the body of the loop is never executed; *got* remains false. Thus, the value of *got* tell us whether or not a *msg* was input from port *l*. Imperative broadcast input asks each child in turn to output a message. As soon as a child is found that will satisfy the request, the polling terminates.

Conditional input, conditional output, and conditional broadcast input are all similar. In each case, the specific child (or each child in succession) is asked whether or not it can supply the requested message. If so, the message is accepted and moved up to the top row of padding processors so that it is available for the parent. The padding tree will have only one instance of a particular message stored in it at a time. If the guard is part of a loop, the message will be re-requested before the next iteration of the loop.

Conditional input and output require the use of both routing numbers, but do not send up an explicit *no* answer. Conditional broadcast input uses the *no* answer to continue polling the children, but needs only the depth finder routing number.

Appendix 1 contains templates for generating skeleton programs for the padding processors. The compiler uses these templates, filling in specific port and message references as needed. The padding processor program is a conditional statement, with a guard for every possible communication between parent and child.

4. Communication Primitives in the Hardware

We devised two criteria for the tree machine processor and port design. First, the design must use chip area sparingly. In general, we are willing to sacrifice execution speed in order to limit the number of functions built into the hardware. Each new instruction is carefully scrutinized before being added to the repertoire. Second, the processor should reflect the structure and scope of TMPL. We intend to program this machine solely in TMPL. Thus the instruction set is designed for easy translation from TMPL to machine code.

These two goals compliment each other when designing instructions to implement TMPL statements that do not involve communication. However, when we attempt to implement directly the rich communication structure of TMPL, we find the complexity of the processor, and thus the area it occupies, increasing.

The double queue implementation discussed below is a nice compromise. It provides direct implementation of two of the four communication modes, conditional input and imperative output. The other two modes can be implemented via compiler translation into the chosen modes. This transformation is discussed later.

4.1. An Overview of the Queue Design

A TMPL port is a bidirectional one: the same port is used for both input and output. The underlying hardware has two distinct unidirectional connections between the processor, controlled by a port. The hardware port is a simple micro-coded processor, distinct from the tree machine processor. The port and its instruction set are described in more detail in the next section.

Each unidirectional connection between processors has an associated queue that buffers messages, thus decoupling the two processors, as shown in Figure 4. The length of the queue determines the independence of the communicating processors, but cannot matter to the algorithms controlling the port. One may assume a length of one for all of the discussion that follows. A queue size will eventually be chosen through experimentation and simulation.

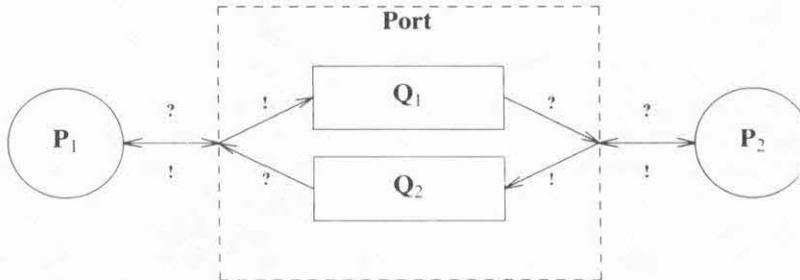


Figure 4. Communication Using Queues.

Each access to a port causes one word in the associated queue to be read or written. Thus, a TMPL message with several arguments is compiled into a sequence of messages, one for each argument:

$!!\text{arc}(i,j)$ becomes $!!\text{arc}; !i; !j$

$!?\text{arc}(i,j) \rightarrow$ becomes $!?\text{arc} \rightarrow !?i; !?j$

Notice that the message name is no longer associated with its arguments. If the two communicating processors have different notions of how many arguments the message contains, chaos will ensue. An alternative implementation would tag each argument with the message name, as in the example below.

$!!\text{arc}(i,j)$ becomes $!!\text{arc}(i); !!\text{arc}(j)$

Tagging provides some measure of run-time validity checking on messages, at the cost of extra bits in each entry in the queue. Since we have complete information about the nature of the communication between processors at *compile* time (see Figure 2), we do better to put the checking there, retaining our streamlined port design.

4.2. The Port Instruction Set

The port responds to three instructions from the processors it interfaces. **MATCH(msg)** compares its argument with the first element in the queue. It returns **true** if they match, **false** if they are different, and **maybe** if the queue is empty. **READ** removes the first element from the queue and returns its value to the requesting processor. If the queue is empty, it waits until something is inserted. **WRITE(msg)** adds its argument to the end of the queue, returning **true** to the writer. If the queue is full, **maybe** is returned, and the writer must retry the operation. Figure 5 formally presents the algorithms. We assume the usual operations on queues (called Q in Figure 5): $\text{empty}(Q)$ is a boolean function that is **true** when the queue is empty, $\text{head}(Q)$ returns the value of the first element in the queue, $\text{removeHead}(Q)$ returns the first element and removes it from the queue, $\text{full}(Q)$ is a boolean function that is **true** when the queue is full, and $\text{add}(Q, \text{msg})$ adds msg to the end of the queue.

It remains to show how TMPL message statements and expressions are compiled into port instructions. We will look only at conditional input and imperative output. The next section describes a technique for transforming the missing communication modes into these.

Conditional inputs require the **MATCH** and **READ** instructions. **MATCH** is used to satisfy the condition: the message will be input only if **MATCH** returns **true**. The arguments are retrieved with **READ** instructions.

```

{ MATCH(msg) →
  [ NOT empty(Q) AND head(Q)=msg → return(TRUE)
  | NOT empty(Q) AND head(Q)≠msg → return(FALSE)
  | empty(Q) → return(MAYBE)
  ]
| READ →
  [ NOT empty(Q) → return(removeHead(Q))
  | empty(Q) → SKIP
  ]
| WRITE(msg) →
  [ NOT full(Q) → add(Q,msg); return(TRUE)
  | full(Q) → return(MAYBE)
  ]
}

```

Figure 5. The Port Instruction Set

$$p?arc(i,j) \rightarrow \text{becomes} \quad MATCH(arc) \rightarrow READ; i:=READ; j:=READ$$

Imperative outputs use only the WRITE instruction, but are complicated by the fact that the output will fail if the queue is full. Thus, they are implemented in a loop, shown in the TMPL notation below.

$$p!arc(i,j) \quad \text{becomes} \quad \begin{array}{l} [WRITE(arc) \rightarrow SKIP]; \\ [WRITE(i) \rightarrow SKIP]; \\ [WRITE(j) \rightarrow SKIP] \end{array}$$

The first implementation of the port utilizes queues of length two. The port is split between the two processors that share it, with one word of each queue in each processor. Transfers between the two halves of the queue is done bit-serially, with one bit transferred during each storage cycle.

4.3. Other Comments about the Processor

The processors in our present design have 12-bit registers, and a 12-bit address for accessing a program store organized as 4-bit nibbles. Instructions are variable length, ranging from two to six nibbles long. Most data is stored in the sixteen general purpose registers. Floating point numbers occupy four registers: one for the exponent, and three for the mantissa. A subroutine for floating point multiply is about 150 nibbles long and executes in about 600 storage cycles.

This relatively serial organization and direct connection to on-chip storage allows a short storage cycle and simple instruction pre-fetch techniques. Internal pipelining in the pre-fetch organization means that some nibbles fetched are not executed but this cycle is used to good advantage to refresh dynamic storage. The processors are small enough that we expect to fit four of them with 1024 nibbles of storage each on a chip with $\lambda=1$ micron, or one per chip with $\lambda=2$ micron. By adhering to the principles of smallness and regularity, the processors achieve very good duty-factors on their internal parts, and are expected to be very fast.

5. Mapping TMPL Primitives onto the Hardware

The hardware design places a pair of queues between the two communicating processors. It supports three operations: putting a message at the end of the output queue, removing the top element of the input queue, and non-destructively examining the top element of the input queue. These operations correspond to imperative output and conditional input. TMPL programs also contain imperative inputs and conditional outputs, and these must be transformed into the two forms

that are implemented. We discuss that mapping here.

5.1. Imperative Input

An imperative input statement is semantically identical to a conditional statement with only one guard, the expression form of the imperative, and a no-op action following the guard;

$$p?msg \equiv [p?msg - SKIP]$$

Message expressions remain in the **maybe** state until there is some activity on the port named in the expression. As long as not all of the guards in a conditional or loop statement are false, the statement will not terminate. In this case, the processor will wait until something arrives at the port. If the message matches, the input is done, the no-op is executed, and the conditional statement is completed. If some other message is pending on the port, the conditional statement aborts, as does a mismatched imperative.

We have just changed an imperative input into a conditional, and in the process, may have created an illegal conditional to conditional communication. If we stopped here, that would be a major concern. However, we are also going to remove all conditional outputs in this compilation step. With only conditional inputs and imperative outputs remaining, it is impossible to have an invalid pairing of conditionals.

5.2. Conditional Output

When conditional output is used, the processor is asking the port whether the other processor is waiting to input the message. The conditional output can be restated as a pair of communications. First the processor issues a *message statement* asking, in essence, if the other processor wants the message. This imperative communication goes *outside* the body of the conditional or loop statement that contains the original conditional output. That expression is replaced with a *conditional input* waiting for a positive reply to the query. The replacement technique is shown below.

$$[p!msg - command] \text{ becomes } p!query ; [p?yes - p!msg; command]$$

Notice that a new message, *query*, has been introduced. The program in the processor at the other end of the port must now be changed to accept the new message. The message typing discussed earlier makes it trivial to identify the imperative input that is paired with the original conditional output, and replace it with a conditional statement that waits for the *query* message, answers it, and then inputs *msg*. The replacement code is given below.

$$p?msg; \text{ becomes } [p?query - p!yes ; [p?msg - SKIP]]$$

The example above uses a conditional statement with a single guard. The same replacement technique is used for statements with multiple guards, but there is a subtle difference in the code in the processor doing the imperative inputs: it must clear *all* of the query messages before requesting the original message. Figure 6 shows a loop statement with three conditional outputs as guards, and an imperative input that matches one of them. We show how the code in both processors is rewritten to remove the conditional outputs from one and the imperative input from the other.

Removing conditional outputs from the source code is not as clean as one might like. The connection plan for the tree must be used to identify the port connections, since both processors involved in the conditional output must be modified. A solution that requires changing only the program in the processor actually doing the conditional output is preferred, but elusive.

6. Conclusions

We have described a strategy for providing local communication among small, simple processors connected like a binary tree. The design has both software and hardware components.

The software, a compiler, allows the programmer to write programs for trees with arbitrary fanout using a rich set of communication primitives. The compiler recursively applies a mapping algorithm that assigns logical ports to physical ones until the tree with arbitrary fanout has been

Original Source	Rewritten Source
<pre> { p!m1 → SKIP p!m2 → SKIP p!m3 → SKIP } </pre>	<pre> p!query1; p!query2; p!query3; { p?ack1 → p!m1; SKIP p?ack2 → p!m2; SKIP p?ack3 → p!m3; SKIP } </pre>
<pre> p?m2; </pre>	<pre> [p?query1 → SKIP]; [p?query2 → [p?query3 → SKIP]; p!ack2; [p?m2 → SKIP]]; </pre>

Figure 6. Removing Conditional Outputs

made binary. Skeleton programs for the intermediate processors that pass messages between parent and child are produced.

The hardware design places a pair of queues between processors. The queues implement a subset of the software primitives; the others can be mapped onto them by the compiler. The advantage of the queue scheme is its simplicity. Because the power of the tree machine comes from the *number* of processors, and not the capabilities of an individual one, a hardware design that is small, simply described, and easy to use, is desired.

Acknowledgements. The research described here was sponsored in part by the Defense Advanced Research Projects Agency, ARPA order #3771, and monitored by the Office of Naval Research under contract #N00014-79-C-0597. Our starting point was a sketch of the machine presented in S. A. Browning's doctoral dissertation.¹ A group of nine people met regularly to refine that sketch into a first implementation in silicon. Martin Rem, Lennart Johnsson, and Peggy Li made valuable contributions to the discussions of mapping problems between the notational abstraction and the machine implementation. Each potential design was examined in light of the requirements of the notation until one was found that satisfied both the programmers and the hardware designers. A compiler like the one described here is being implemented by S. A. Browning at Bell Laboratories.

A tree machine processor has been designed and is being laid out by C. L. Seitz, Howard Derby, Chris Kingsley, and Chris Lutz. Erik deBenedictis and Peggy Li have written a collection of programs to evaluate the processor design.

References

1. Sally A. Browning, *The Tree Machine: A Highly Concurrent Computing Environment*, California Institute of Technology (1980). Computer Science Technical Report #3760
2. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey (1976).
3. C. A. R. Hoare, "Communicating Sequential Processes," *C.ACM* **21**(8), pp. 666-677 (August, 1978).
4. Charles L. Seitz, "Self-Timed VLSI Systems," *Proc. Caltech Conference on Very Large Scale Integration*, pp. 345-356 (January, 1979).
5. Ivan E. Sutherland and Carver A. Mead, "Microelectronics and Computer Science," *Scientific American* **237**(3), pp. 210-228 (September, 1977).

Appendix 1. Templates for Padding Programs

1. Imperative output: $c(i)!$ msg

```

parent:
  [ odd(i) → !msg((n-3)/2,(i+1)/2)
  | even(i) →
    [ n=3 → r!msg
    | n>3 → r!msg((n-4)/2,i/2)
    ]
  ]

padding:
  p?msg(d,p) →
  [ d=0 AND odd(p) → !msg
  | d<2 AND even(p) → r!msg
  | d>0 AND odd(p) → !msg((d-1)/2,(p+1)/2)
  | d>1 AND even(p) → r!msg((d-2)/2,p/2)
  ]

```

2. Imperative Broadcast Output: $c(*)!$ msg

```

parent:  l,r!msg

padding:  p?msg → l,r!msg

```

3. Imperative Input: $c(i)?$ msg

```

parent:
  [ odd(i) →
    !request((n-3)/2,(i+1)/2) ;
    l?msg
  | even(i) →
    [ n=3 → SKIP
    | n>3 → r!request((n-4)/2,i/2)
    ] ;
    r?msg
  ]

padding:
  p?msg(d,p) →
  [ d=0 AND odd(p) → l?msg
  | d<2 AND even(p) → r?msg
  | d>0 AND odd(p) →
    !request((d-1)/2,(p+1)/2) ;
    l?msg
  | d>1 AND even(p) →
    r!request((d-2)/2,p/2) ;
    r?msg
  ] ;
  p!msg

```

4. Imperative Broadcast Input: $c(*)?msg$

parent:

```

!req((n-3)/2);
[ !?msg → SKIP
 | !?no →
   [ n=3 → SKIP
   | n>3 → r!req((n-4)/2)
   ] ;
r?msg
]

```

padding:

BOOLEAN got;

p?req(d) →

got:=FALSE ;

[d=0 →

{ NOT got AND !?msg → got:=TRUE }

| d>0 →

!req((d-1)/2);

[!?msg → got:=TRUE

| !?no → SKIP

]

];

[NOT got AND d<2 →

{ NOT got AND r?msg → got:=TRUE }

| NOT got AND d>1 →

r?req((d-2)/2);

[r?msg → got:=TRUE

| r?no → SKIP

]

];

[got → p!msg

| NOT got → p!no

]

5. Conditional Output: $c(i)!msg \rightarrow$

parent:

```
[ odd(i) → !msg((n-3)/2,(i+1)/2)
| even(i) →
  [ n=3 → SKIP
  | r>3 → r!msg((n-4)/2,i/2)
  ]
]
```

l,r?yes →

```
whatever was here
[ odd(i) → !msg((n-3)/2,(i+1)/2)
| even(i) →
  [ n=3 → SKIP
  | r>3 → r!msg((n-4)/2,i/2)
  ]
]
```

padding:

BOOLEAN got;

```
| p?msg(d,p) →
  [ d=0 AND odd(p) →
    got := FALSE ;
    { NOT got AND !msg → got:=TRUE };
    [ got → p!yes
    | NOT got → SKIP
    ]
  | d<2 AND even(p) →
    got := FALSE ;
    { NOT got AND r!msg → got:=TRUE };
    [ got → p!yes
    | NOT got → SKIP
    ]
  | d>0 AND odd(p) → !msg((d-1)/2,(p+1)/2)
  | d>1 AND even(p) → r!msg((d-2)/2,p/2)
  ]
| l,r!yes → p!yes
```

6. Conditional Input: $c(i)?msg \rightarrow$

parent:

```
[ odd(i)  $\rightarrow$  l?req((n-3)/2,(i+1)/2)
  | even(i)  $\rightarrow$ 
    [ n=3  $\rightarrow$  SKIP
      | r>3  $\rightarrow$  r!msg((n-4)/2,i/2)
    ]
  ]
```

l,r?msg \rightarrow

```
whatever was there
[ odd(i)  $\rightarrow$  l?req((n-3)/2,(i+1)/2)
  | even(i)  $\rightarrow$ 
    [ n=3  $\rightarrow$  SKIP
      | r>3  $\rightarrow$  r!msg((n-4)/2,i/2)
    ]
  ]
```

padding:

BOOLEAN got;

```
| p?req(d,p)  $\rightarrow$ 
  [ d=0 AND odd(p)  $\rightarrow$ 
    got:=FALSE ;
    { NOT got AND l?msg  $\rightarrow$  got:=TRUE } ;
    [ got  $\rightarrow$  p!msg
      | NOT got  $\rightarrow$  SKIP
    ]
  | d<2 AND even(p)  $\rightarrow$ 
    got:=FALSE ;
    { NOT got AND r?msg  $\rightarrow$  got:=TRUE } ;
    [ got  $\rightarrow$  p!msg
      | NOT got  $\rightarrow$  SKIP
    ]
  | d>0 AND odd(p)  $\rightarrow$  l!req((d-1)/2,(p+1)/2)
  | d>1 AND even(p)  $\rightarrow$  r!req((d-2)/2,i/2)
  ]
| l,r?msg  $\rightarrow$  p!msg
```

7. Conditional Broadcast Input: c(*)?msg →

parent:

```

!!req((n-3)/2) ;
[ l?yes → SKIP
| l?no →
  [ n=3 → SKIP
  | n>3 → r!req((n-4)/2)
  ] ;
[n=3 → SKIP
| n>3 AND r?yes → SKIP
| n>3 AND r?no → SKIP
]
]

```

l,r?msg →

```

whatever was there
!!req((n-3)/2) ;
[ l?yes → SKIP
| l?no →
  [ n=3 → SKIP
  | n>3 → r!req((n-4)/2)
  ] ;
[ r?yes → SKIP
| r?no → SKIP
]
]

```

padding:

BOOLEAN got;

p?req(d) →

```

got:=FALSE ;
[ d=0 →
  { NOT got AND l?msg → got:=TRUE }
| d>0 →
  !!req((d-1)/2);
  [l?yes → got:=TRUE ; l?msg
  | l?no → SKIP
  ]
] ;
[ NOT got AND d<2 →
  { NOT got AND r?msg → got:= TRUE }
| NOT got AND d>1 →
  r?req((d-2)/2);
  [r?yes → got:=TRUE ; r?msg
  | r?no → SKIP
  ]
] ;
[got → p!yes ; p!msg
| NOT got → p!no
]

```