BBN Systems and Technologies Corporation
A Subsidiary of Bolt Beranek and Newman Inc.

-> f.le: Monarch

BBN-TIR 113

The Monarch Parallel Processor Design

Randall D. Rettberg
William R. Crowther
Philip P. Carvey
Raymond S. Tomlinson

17 January 1989

The Monarch Parallel Processor Design

Randall D. Rettberg
William R. Crowther
Philip P. Carvey
Raymond S. Tomlinson

BBN Systems and Technologies Corporation

17 January 1989

The Monarch Parallel Processor Design

Table of Contents

The Monarch Parallel Processor Design

Randall D. Rettberg
William R. Crowther
Philip P. Carvey
Raymond S. Tomlinson

BBN Systems and Technologies Corporation

17 January 1989

Abstract

The Monarch parallel processor takes advantage of custom VLSI in the design of a shared-memory parallel processor. Monarch configurations have been examined with as many as 65,536 processors, each processor having equal access to a 65,536-port memory system of 16 gigabytes. Such a machine would have a peak execution rate of 390 billion instructions per second or 130 billion floating point operations per second. The network that connects the processors to the memories has 16% contention and supports message combining to reduce the effect of hot spots. Processors use a full/empty tag on every word of memory for synchronization and the shared memory for communication. This simple structure eases the task of programming such a massively parallel machine.

1. Introduction

The recent flurry of parallel processing companies and new parallel machines is the result of a simple technical observation: interesting computer systems can be made by lashing together many individual computers. If these computers can be made to work together, the resulting system will provide more computing power than can be achieved in any other way today. Furthermore, if the architecture allows, machines of varying size and power can be built from the same design; and, if some of the processors fail, the others should be able to continue to provide service.

The rapid advance of semiconductor technology supports this notion. In the days of ENIAC and Whirlwind, when one processor filled a room with vacuum tubes, it was hard

to consider lashing many of them together. The minicomputer, a computer in a single cabinet, enabled the first early parallel machines to be built (C.mmp [Fuller 78] [Wulf 78], Cm* [Jones 80], Pluribus [Heart 73]). However, only the arrival of the microprocessor has allowed serious consideration of machines with hundreds or thousands of processors.

Since then, semiconductor technology has become more accessible. In 1980, Carver Mead and Lynn Conway published *Introduction to VLSI Systems*. They made this prediction:

> VLSI electronics presents a challenge, not only to those involved in the development of fabrication technology, but also to computer scientists and computer architects. The ways in which digital systems are structured, the procedures used to design them, the trade-offs between hardware and software, and the design of computational algorithms will all be greatly affected by the coming changes in integrated electronics. We believe this will be a major area of activity in computer science on through the 1980s.

The technology of multiproject chips and a silicon foundry (the MOSIS system) [Cohen 81] operated by DARPA allowed BBN researchers and engineers to consider the design of a parallel processor as a complete system from silicon to software.

At BBN, we saw this technological opportunity in the context of a decade of experience in distributed systems and parallel computers. To us, the choice of the machine architecture was clear.

We distinguish four primary categories of parallel architecture: data parallel (Connection Machine [Hillis 85]), systolic [Computer 87] (WARP [Kung 78]), message passing (cubes [Seitz 85]), and shared memory (Encore, Butterfly™ [BBN 87]). While each architecture has its own proponents, the shared-memory structure has been most appealing to us and has been the focus of our machine designs.

We define a shared-memory machine as a parallel computer in which all of the processing elements operate independently and are connected to a unified memory system. While different researchers have different definitions for the various categories of architecture, in our view, an ideal implementation of a shared memory machine would have the following properties:

- All of the processors have equal access to all of the memory locations. The physical address of a memory location is the same from all processors, and the delay in referencing a memory location is small and is the same from all processors.

- Processors see no additional delays in their memory references due to references made by other processors. That is, there is no contention for memory and there are no conflicts in getting to memory.
- Communications between processors and synchronization of the processors are accomplished through the shared memory.

BBN has built two machines with these properties in mind. The first, the Pluribus [Heart 73], used the Lockeed Sue minicomputer and a bus-based distributed crossbar switch to connect processors to a central memory. The largest Pluribus was constructed in 1974 and had 14 processors. In all, we built and delivered about fifty Pluribus machines.

The second machine is the Butterfly™ parallel processor. Based on the Motorola 68000 (and later the 68020), it uses a multistage interconnection network called the Butterfly switch, to allow the processors direct access to each other's memory. In 1985, BBN constructed a 256-processor Butterfly system. We have delivered approximately 100 Butterfly systems, and are developing a technology upgrade to increase the performance of Butterfly systems.

In 1984, under support of the DARPA Strategic Computing Program, BBN began the design of a new machine architecture, the Monarch parallel processor, which allows for large-scale parallelism while approaching the ideal of shared memory to a greater degree than ever before. In this design, we have taken the view that it is not necessary to compromise on the communications between the processors and their data — all processors can have efficient access to the whole of the system memory. This one concept chiefly distinguishes the Monarch architecture from others and makes it relatively straightforward to program.

Today, the design of the Monarch is largely done and we are well along in the implementation. Components of the interconnection network of the machine have been fabricated and are now under test. The logic design of the processor is almost complete and ready to begin the implementation. We have analyzed the software in detail through hand-coded examples, a simulator, and a rudimentary compiler.

The Monarch does not yet exist as a machine. Nonetheless, we will use the present tense in the rest of this paper, saying "the Monarch does so and so" rather than the awkward but more accurate "the Monarch is designed to do so and so and the implementation looks good." Please excuse this inaccuracy.

2. Hardware Description

The Monarch architecture may be implemented in many different configurations. Several examples are shown in Table 1.

Table 1 — Capacities of Various Monarch Configurations

Processors	Peak MIPS	Peak MFLOPS	Memory Size (Megabytes)*	Memory Bandwidth (MBytes / Sec)
1	6	2	0.5	8
1,024	6,144	2,048	268	8,192
8,192	49,152	16,384	2,147	65,536
65,536	393,216	131,072	17,180	524,288

* Assuming 1Megabit DRAMs

Each Monarch processor operates at a peak rate of 6 MIPS or 2 MFLOPS. A Monarch system with 65,536 processors operating together would have a peak execution rate of over 390 billion instructions per second or over 130 billion 64-bit floating point operations per second. One memory module contains 0.5 megabyte of memory and can support 2 processors; thus the 65,536-processor machine has 17 gigabytes of physical memory.

The logical structure of the Monarch is shown in Figure 1. A large number of processors access a large number of memory modules through an interconnection network. Peripheral devices are connected to some of the processors. In addition, there are systems for clock and power distribution, for packaging and cooling, and a special system for maintenance, diagnosis, and operation of the hardware.

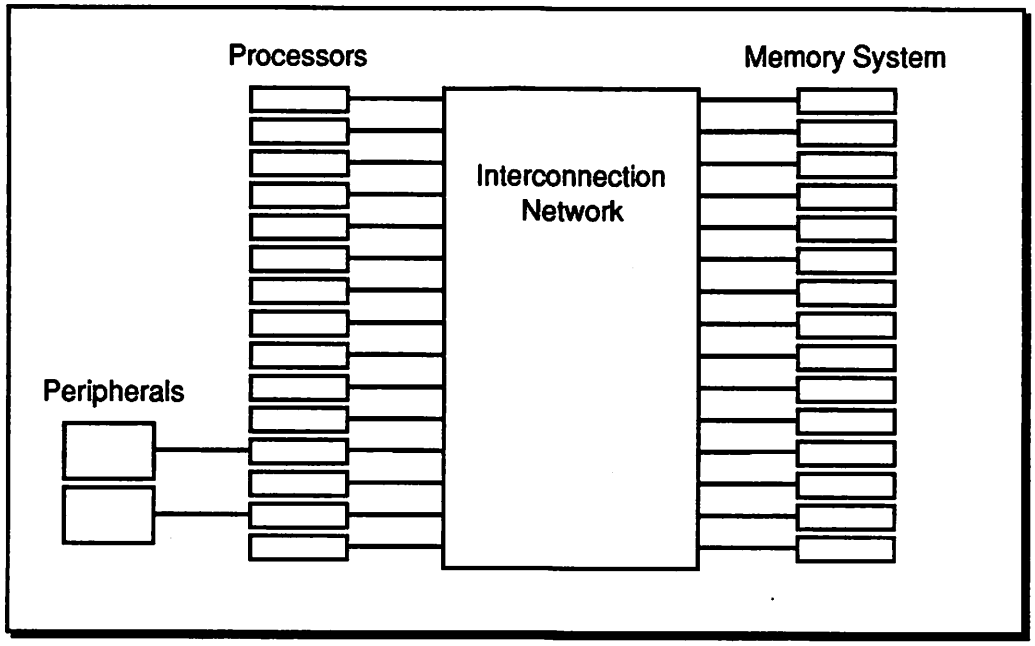

Figure 1 — Monarch System Design

The Monarch is a 64-bit machine with tags. Each location in memory and each register in the processor has 64 bits of data and 8 bits of tag. The Monarch processor is a single custom CMOS device. It includes hardware for floating point and a memory management unit. Virtual addresses are 40 bits wide, allowing access to a trillion-byte paged address space. A small local memory (three DRAMS) is attached to each processor for instruction storage. We think of this as a poor man's instruction cache since it must be loaded by the application or operating system software.

The memory module consists of a custom memory controller and ten commercial DRAMs. One memory module provides 512 kilobytes of dual-ported, tagged, 64-bit word memory with full error detection and correction.

The software sees a machine having thousands of identical processors all with equal access to a single memory system. This is achieved by a high-performance interconnection network that routes memory references from the processors to the memory modules in a series of stages from switch node to switch node based on an address in the memory reference. The switch is designed to operate at a clock rate of 175 MHz and a bit rate of 350 MHz. This Monarch network is based on the Butterfly switch [Rettberg 86].

Even with a conflict-free interconnection network, if many processors access as many memories with a random access pattern, a machine structure like the Monarch's would be expected to experience a contention rate of about 36% due to conflicts at the memory.

However, by providing more than one path through the network and more than one port to each memory location, we expect the total contention to be only about 16% for even the largest machine, and even when every processor is referencing memory at its maximum rate. There is enough network and memory bandwidth to allow every processor to access memory at every opportunity.

This high level of network performance is attained with only a modest investment in the network. In the largest machine, custom devices in the network account for only about 35% of the custom ICs and only 11% of the total ICs in the machine. The Monarch interconnection network uses just two types of custom CMOS switching elements, a switch and a statistical concentrator.

Four facilities are included in the Monarch design specifically to support parallel processing:

1. The memory system permits a processor to *steal* a memory location, thus keeping other processors from accessing it. This is the basic synchronization primitive.
2. The interconnection network spreads consecutive memory locations across the memory modules. This effectively *interleaves* the memory system to reduce systematic memory contention.
3. *Read combining* in the interconnection network and memory controller allows any number of processors to read the same location simultaneously. *Read combining* eliminates the primary cause of memory hot spots — individual hot variables.
4. A processor may reduce its load on the system when it is "polling" a variable by using a *low priority read.* Lower priority is achieved by limiting low priority read requests to be sent only once every few microseconds, while normal traffic can be sent every microsecond.

Monarch processors may operate independently or in concert as desired. Each processor may have its own instruction stream and its own data set, in which case the Monarch acts as thousands of separate computers. However, the processors may share a set of data and a single program applying parallel processing power to a single problem. Even in this case, different processors will frequently be executing at different points in the instruction stream and accessing different elements of the data set.

Through a Mach version of the UNIX® operating system (developed at Carnegie-Mellon University), a number of these processors are allocated to each user in a form of *space-sharing* that augments the conventional *time-sharing* of processor resources.

The Monarch was born of an exploration of high-speed VLSI design and a desire to build a better shared-memory parallel processor. We are now in the middle of the significant effort required to implement the custom CMOS designs. However, because we kept the structure of the machine simple, we have only four chips to design (processor, memory controller, switch, and concentrator). The computing power of the machine will result not from an elaborate architecture, but from the mass production of moderate components. This, we hope, will allow our small group to develop a very powerful machine.

The Monarch effort has been underway for about four years. During that time, we have refined the design and studied the software implications of our choices. We have worked through a few dozen different example programs, simulated their performance, and analyzed their behavior. This experience satisfies us that, when completed, the Monarch will be a usable and pleasant machine to deal with, as parallel processors go.

2.1 Monarch Interconnection Network

The Monarch's interconnection network is key to the success and viability of the machine. As a framework for the rest of the machine, it defines a high-speed serial signaling strategy, a system wide time frame, and rigid message formats. While the Monarch interconnection network is unique, it is a member of the family of multistage interconnection networks that includes the Butterfly switch, the banyan, and the omega networks [Broomell 83]. Similar switches are used in the RP3 [Pfister 87] and the Ultracomputer [Gottleib 87].

Monarch interconnection networks can be designed in a variety of configurations — the best structure depends on the detailed characteristics of the machine they support. Larger machines need more switch ports and more switch bandwidth. The Monarch network can be configured to meet these needs almost without limit.

The Monarch operates in a simple and synchronous manner. At the beginning of every microsecond, the processors send memory reference messages into the network. The network routes the messages to the memory. The memories then send reply messages back through the network to the originating processors. This memory referencing cycle, called a *frame*, forms one of the basic measures of time in the machine. Each processor is able to perform one memory reference during every frame.

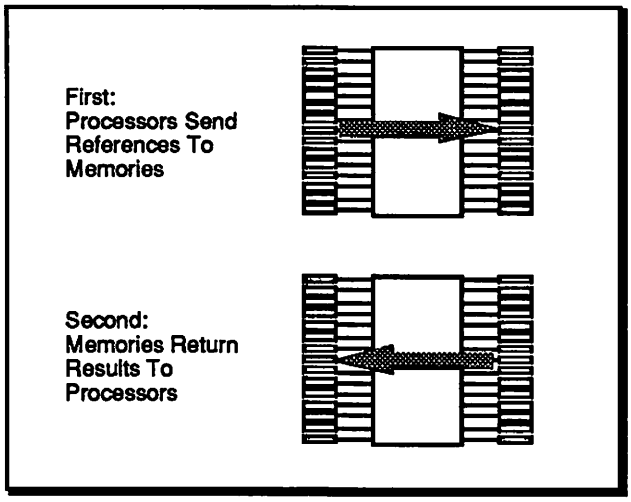

Figure 2 – Synchronous Operation

Data flows through the network serially along one-bit-wide channels at very high speeds. A technique called *dynamic delay adjustment* automatically compensates for clock and data skews. Special driver and receiver circuits allow the custom CMOS devices to communicate at data rates of 350 megabits per second.

In the following sections, we describe the principles behind these networks, provide a simple mathematical analysis of their performance, and address some of the interesting implementation details.

Monarch Network Topologies

Monarch interconnection networks are highly variable in their configuration. Not only can the number of inputs and outputs be varied, but the properties of contention and delay can be adjusted as well. Across these designs, the principles remain the same. Figure 3 illustrates the operation of the Monarch using a network that is very small by Monarch standards. It has only 32 input and 32 output ports.

In this figure, each processor is represented by the letter *P* and each memory module by the letter *M*. There are two types of switching elements: *switches* and *concentrators*. Switches route messages through the network interconnect based on an address contained in the first few bits of the incoming message. In this example, the first three bits of the message specify one of eight possible output ports. Each output port has two output channels. If either is free, the message moves forward along that channel; otherwise, it is discarded and the sending processor must try again. The bits used to specify the output port are removed from the message.

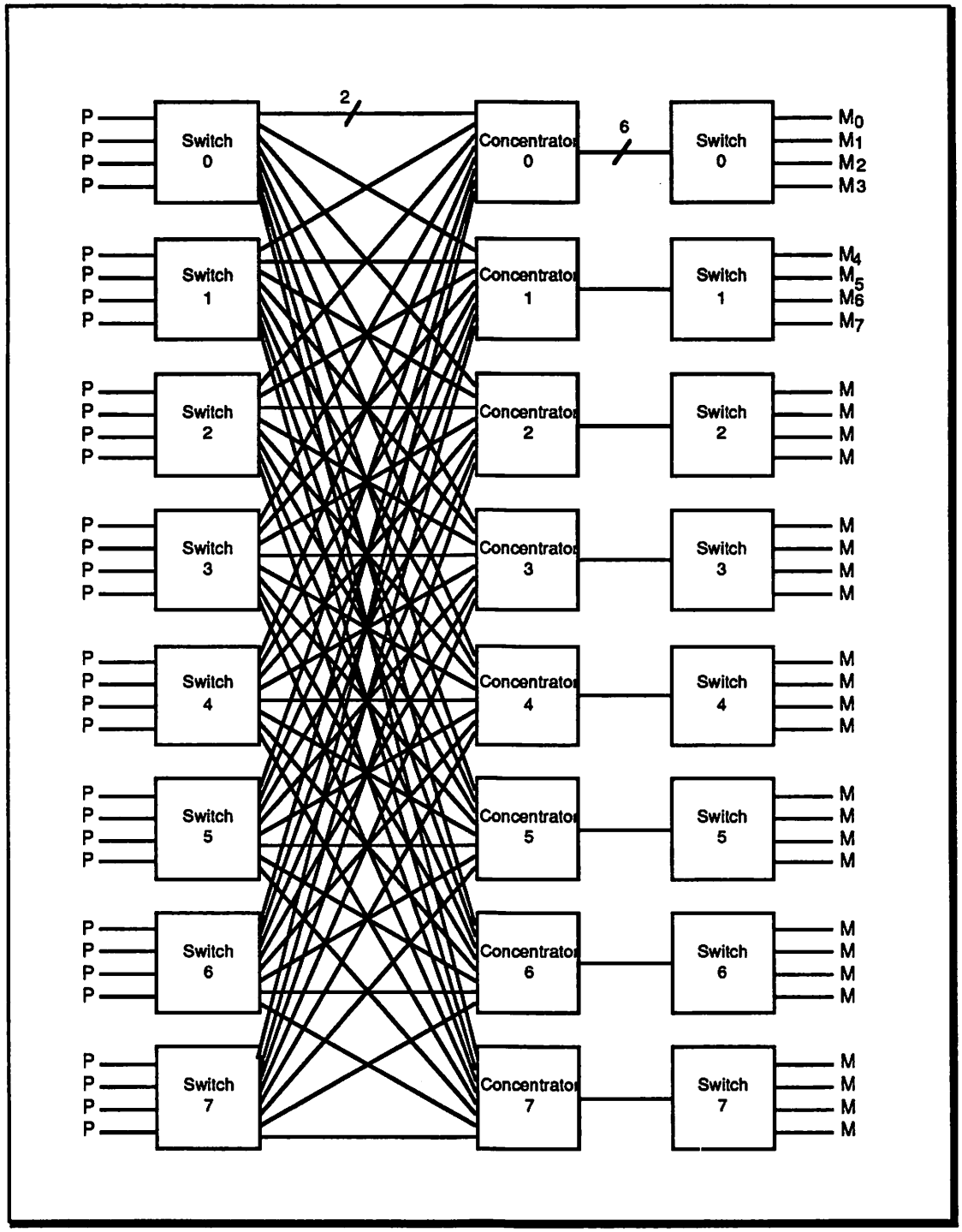

Figure 3 – 32 Processor Monarch

Concentrators acts as statistical multiplexors. A concentrator allocates arriving messages to one of its output channels; if no output is free, it discards the message. Concentrators perform no routing functions for the network since all six outputs are connected to the same

destination in the next column of the switch. The purpose of the concentrator is to reduce the number of wires presented to a later stage of the network by accepting many lightly loaded wires and producing fewer more heavily loaded wires.

A Monarch network operates stage by stage. Imagine that processor 0 wants to access a location in memory module 7. It generates a message with a routing header of $\{1, 3, \cdots\}$. The first switch sees the first digit, the 1, and routes the message to concentrator 1 using one of the two available channels. It removes the used routing digit as it forwards the message. Concentrator 1 has an output line available and passes the message on to switch 1 in the third column. Switch 1 sees the digit 3, and passes the message on to the desired memory module. The response to this request returns along the same path but in the opposite direction, from switch 1 to concentrator 1, to switch 0, and finally to processor 0.

Other processors use the network to access memory at the same time. Were they to present the same message to the network, they would access the same memory location — memory addresses are uniform across the machine.

Larger Monarch networks have more stages of switches and concentrators and they become much harder to draw. As an example, a few rows of a 65,536-processor machine are illustrated in Figure 4.

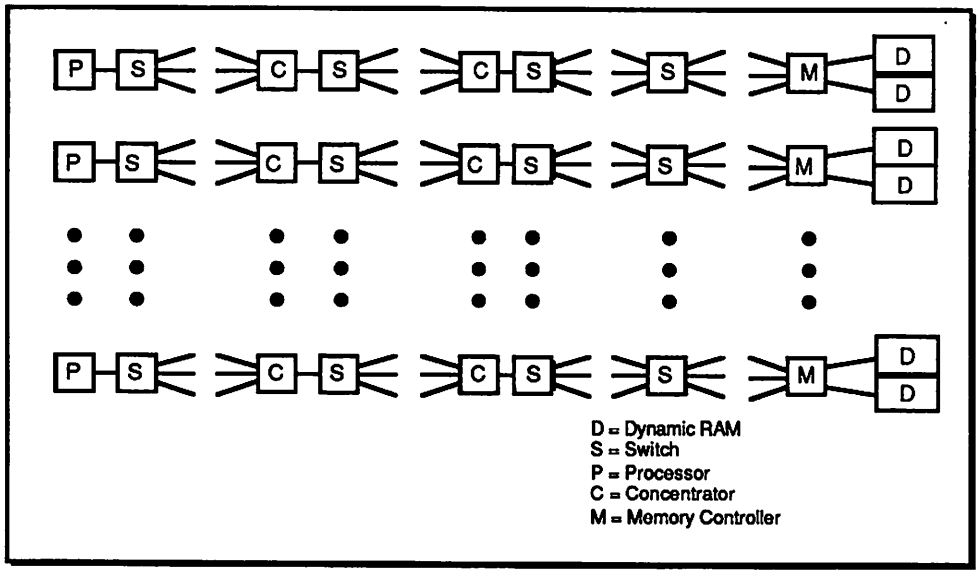

Figure 4 — Switching Structure of a 65,536-Processor Machine

The number of columns in the network is logarithmic in the number of inputs (**n**), and the complexity of the interconnection network is $O(n \log_b n)$. The base of the logarithm (**b**) is the number of output ports on the switch chips. By increasing **b** to the largest value which is still cheap to implement with custom VLSI we minimize the size of the interconnect. A switch with 12 inputs and 16 pairs of outputs lets us efficiently construct machines ranging in size from 100 to 65,536 processors.

Network Operation

The limiting resource in the machine is the bandwidth of the interconnection network. It determines the rate at which the processors can update the memory and the execution speed of the machine. Network bandwidth in turn is determined largely by the product of the number of wires we can run (a mechanical issue) and the speed of an individual wire (an electrical issue).

Knowing that the inter-chip signalling speed would determine the overall performance of future machines, we concentrated on techniques for high-speed signalling between CMOS chips in our early VLSI efforts. We developed line drivers and receivers that operate with lower than normal voltage swings and designed on-chip termination resistors that are digitally adjustable and switchable to reduce power consumption. Along with the use of domino logic and other advanced IC design techniques, the result is communication at data rates of hundreds of megabits per second on each wire.

At high speeds and in large machines, signal and clock skew limit the signalling rates. This limitation requires supercomputer manufacturers to design within strict signal length constraints and even to manually adjust the length of wires and traces on the circuit boards. We developed a different approach, made practical by custom VLSI. Using a patented technique, the custom chips do a kind of electronic "wire trimming" automatically at every input so that variations in signal or clock delay are correctly compensated.

An example of circuitry for *dynamic delay adjustment* is illustrated in Figure 5. In this simplified example, data received by the input pad is sampled by one of five delayed versions of the clock. A data selector is used to select the correct clock. Circuitry similar to this is present at every interchip data pad in the machine. Other circuitry adjusts the clock delay line and chooses the correct setting for the data selectors.

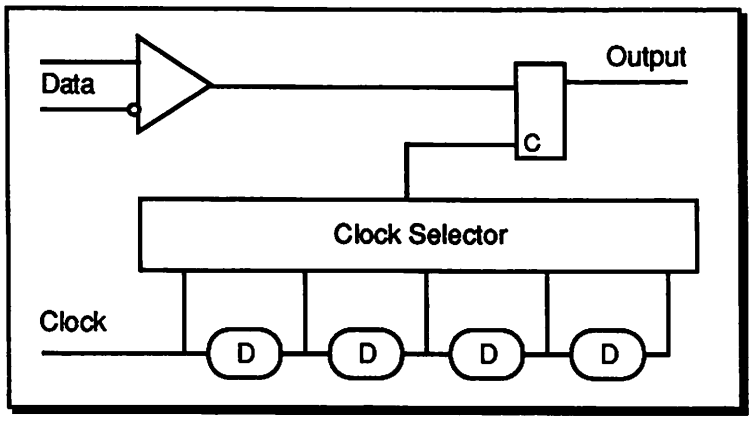

Figure 5 — Dynamic Delay Adjustment

The circuit that select the correct clock phase is shared by all of the pads, providing adjustment for one pad at a time. That circuit uses a synchronization pattern that is sent between chips during an otherwise idle portion of the frame. By sampling the synchronization pattern with different clock phases, it can find the data transitions and select a clock phase that samples in the middle of the data bits. The adjustment circuitry operates slowly enough that metastability is not a problem.

The interconnection network is a cross between circuit-switching and packet-switching. Messages are routed as they arrive based on address bits in their headers, but there is no message storage in the network. Messages are pipelined through the network so that the first bits of a message leave the network before the final bits have even entered the network. This makes efficient use of the wires in the machine, keeps the switch nodes simple, and reduces the delay in accessing memory.

So far, we have depicted the Monarch as having a single network interconnect that joins 65,536 processors to 32,768 memory modules. We have also described a timing regime under which processors always start sending a memory access request at the beginning of a frame. In reality, the dynamic memory chips are fast enough to support two memory references per frame, so we have divided the processors into two groups (called phase A and phase B) and staggered their memory references so that the memory modules can serve each group in turn. Two separate interconnection networks serve each half of the processors. From a software viewpoint, this time multiplexing of the memory system is essentially invisible. From an analytic viewpoint, independence of the two processor groups allows modeling the Monarch as two separate machines, each with 32,768 processors and 32,768 memory modules, joined by a 32,768-port interconnect.

Contention and Hot Spots

Most of the time, a memory reference finds the path to memory free. But occasionally, an output channel is not available because too many other processors are trying to access the same memory module, or they are using the same path through the network. When this happens, messages are ignored (discarded) by the network, leaving processors to retransmit the message later. This is a classic problem in parallel processor design — contention for shared resources. Whenever two processors contend for a single resource, one must wait while the other proceeds. The overall effect of contention translates directly into a statistical slowdown of the machine.

The Monarch is designed to minimize the impact of contention on the performance of the machine. In the next section, we will present an analysis of the contention due to random traffic patterns. We will see that by providing multiple paths in the network and increasing the number of outputs a switching element has, we can provide a low level of contention for random traffic. However, actual traffic will not be random.

Perhaps the most widely known non-random network traffic occurs when many processors access a single piece of data or a small array. This is known as a memory *hot-spot*. Hot spot contention has been studied for networks similar to the Monarch's [Pfister 85] [Thomas 86]. Those studies have shown that references to memory hot spots cause blockage in the network known as *tree saturation,* where not only the hot spot references are delayed, but other unrelated references are delayed as well. A switch that combines references is suggested as the solution to hot spots.

The Monarch switch implements combining, by giving special treatment to read requests in the interconnection network and at each memory module. When two read requests access the same memory location, one proceeds while the other is discarded, but not forgotten. When the reply arrives it is copied back to both requesters.

For example, if 8000 processors are accessing the same 1000-element array, then on the average 8 processors will be accessing each element. At any point in the switching network where there are not enough output wires to support all the transactions simultaneously, the switch chip will detect references to the same memory location and deliver the reply to both sources. As a result of this *read combining*, it is possible for all of the processors in the machine to read the same location in one memory cycle and all get the same result.

Combining switches have previously been very expensive to build (by a factor of 6 to 32 over other switches [Gottleib 87]) because messages were not synchronized; they could arrive at the inputs of a switch at any time. Storage was required to hold the messages so that they could be examined and combined.

Because the Monarch operates in a synchronized serial fashion, the switches, concentrators, and memory controllers are able to combine read requests by comparing the messages bit by bit as they arrive. This serial operation makes it possible to implement a combining switch with 12 input ports and 16 pairs of outputs on a single IC.

In addition to read combining, the Monarch network interleaves memory addresses. If word zero of the memory address space is in memory module zero, the next word (word one) is in the next memory module (memory module one), and so on. References to small data structures or arrays are thus spread across many memory modules, reducing the likelihood that a single memory will have to support all of the traffic.

A third aspect of the design that reduces the effect of hot spots concerns the treatment of messages that fail to get through the network. Instead of holding them in the network, the network discards them and they must compete for switch resources with all other incoming messages during the next frame. In this way, the network is fair and random on every frame.

Interconnection Network Analysis

This section introduces a model of the machine that allows a mathematical analysis of the effect of background contention. The analysis shows that background contention (including both interconnect and memory contention) reduces performance by 16% for the network interconnect topology used in a 65,536-processor Monarch. We have performed a set of experiments using our Monarch simulator software, which closely mimics the topology and the arbitration algorithms of the interconnect and memory system hardware. The results of the experiments verified this modeling technique.

The model makes these assumptions about the operation of the processors, interconnect, and memory:

- Arbitration between requests for resources in the network interconnect is perfectly fair. (The chips use pseudorandom sequence generators to arbitrate conflicting requests.)

- Every processor accesses a random location in main memory once per frame.
- The location addressed by each processor in a particular frame is independent of any memory accesses made during previous frames. In particular, if a memory request is not serviced during the current frame, the processor does not try the same address again on the next frame.
- At each stage of the network, all of the wires connecting a column of switching elements to the next column are statistically identical and independent. (Our simulations model the actual wiring of the network.)

All of the network interconnect topologies discussed in this section can be built from the generalized switch element shown in Figure 6. An incoming memory reference arrives at one of the **a** *inputs* with probability $\mathbf{P_a}$ — this equals the average load on an input. The incoming message contains an address that determines which of the **b** *exit ports* it will be forwarded through. A message can use any one of the **c** *output channels* of the selected exit port. The probability that an output channel carries a message is $\mathbf{P_c}$.

This model applies to the switch chips (*e.g.* **a**=4, **b**=8, and **c**=2 or **a**=6, **b**=4, and **c**=2), the concentrator chips (**a**=16, **b**=1, and **c**=6), and the entire main memory subsystem (**a**=32, **b**=32, and **c**=2) of Figure 3.

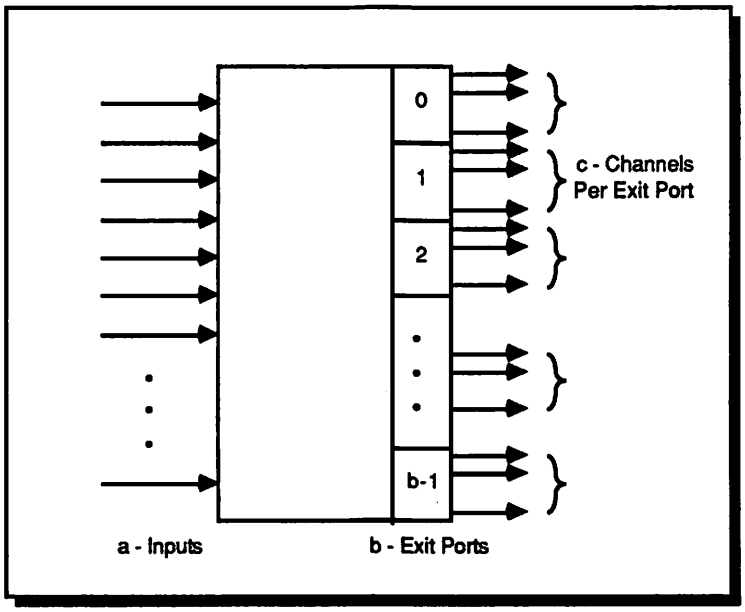

Figure 6 — Generic Switching Element

Assuming that all of the inputs are statistically independent and identical, we can calculate the probability that an output channel is occupied, $\mathbf{P_c}$, for any switch configuration. First,

we calculate the probability, **P{k}**, that exactly **k** messages are routed to one selected output port. Then we calculate the likelihood that a message will use a selected output channel given that **k** messages are present at the output port. We then sum over all possible values of **k**.

P{k} is given by the binomial probability mass function for the number of arrivals in a Bernoulli process where the probability of an arrival is $\frac{P_a}{b}$, the likelihood that an input has a message and that it is routed to a selected one of **b** output ports.

$$P\{k\} = \left(\frac{a!}{(a-k)!\,k!}\right)\left(\frac{P_a}{b}\right)^k\left(1 - \frac{P_a}{b}\right)^{a-k} \qquad \text{(Equation 1)}$$

To get the probability that one of the output channels is occupied, P_c, we must sum the probability that **k** requests arrive at an output port and that a selected output channel will be used if **k** messages do arrive. For example, if there are two output ports, and one message arrives, the likelihood that a selected channel is used is 1/2; if three messages arrive, it is certain that the output channel will be used. P_c is given as follows:

$$P_c = \frac{1}{c}P\{1\} + \frac{2}{c}P\{2\} + \cdots + \frac{c-1}{c}P\{c-1\} + P\{c \text{ or more}\}$$

The probability of **c** or more messages arriving is just 1 minus the probability that fewer than **c** messages arrive.

$$P\{c \text{ or more}\} = 1 - P\{0\} - P\{1\} - P\{2\} - \cdots - P\{c-1\}, \text{ and}$$

$$P_c = 1 - P\{0\} - \frac{c-1}{c}P\{1\} - \frac{c-2}{c}P\{2\} - \cdots - \frac{1}{c}P\{c\} \qquad \text{(Equation 2)}$$

Inserting Equation 1 into Equation 2, we have the solution, the probability that an output channel is active for any switch element configuration given that the inputs are statistically independent and identical.

$$P_c = 1 - \left(1-\frac{P_a}{b}\right)^a - \left(\frac{c-1}{c}\right)\left(\frac{P_a}{b}\right)\left(1-\frac{P_a}{b}\right)^{a-1} - \left(\frac{c-2}{c}\right)(a)\left(\frac{P_a}{b}\right)^2\left(1-\frac{P_a}{b}\right)^{a-2} -$$

$$\cdots - \left(\frac{1}{c}\right)\left(\frac{a!}{(a-c)!\,c!}\right)\left(\frac{P_a}{b}\right)^c\left(1-\frac{P_a}{b}\right)^{a-c} \qquad \text{(Equation 3)}$$

For c = 1 and 2, the equations are simply:

$$P_c\{c=1\} = 1 - \left(1-\frac{P_a}{b}\right)^a \qquad \text{(Equation 4)}$$

$$P_c\{c=2\} = 1 - \left(1-\frac{P_a}{b}\right)^a - \frac{a}{2}\left(\frac{P_a}{b}\right)\left(1-\frac{P_a}{b}\right)^{a-1} \quad \text{(Equation 5)}$$

Having derived these probabilities, we can relate these equations to the macro properties of the system. The expected number of input messages is $\mathbf{a}\,\mathbf{P_a}$. The expected number of output messages is $\mathbf{b}\,\mathbf{c}\,\mathbf{P_c}$, and the efficiency of the switching element is $\frac{\mathbf{bcP_c}}{\mathbf{aP_a}}$.

We can now see several interesting properties of these switching networks. Consider a hypothetical implementation of a large Monarch that uses a full crossbar for its switching network by setting **a**=**b**=32,768. If each memory module has one input port, the throughput formula for **c**=1 applies (Equation 4).

Assume that a message is presented at every switch input during every frame ($\mathbf{P_a}$ is 1). In that case, the efficiency is 63%, or to put it another way, 37% of the messages are lost. (Note that this approaches 1/e.) This is a serious problem, for even with no conflicts in the interconnection network, the conflicts at the memories would waste 37% of the machine!

A solution to this is to provide many more memory modules than processors — e.g., consider using a factor of 2 more memories. Then, **a** = 32,768, **b** = 65,536, **c** = 1, and the performance improves to 79% efficient (21% are lost). This is better, but not much better.

The approach we have taken in the Monarch is to have as many memory modules as processors, but to give each memory module two ports into the switch and design the memory module so that it can service two requests at the same time. Then, from Equation 5, (**a** = 32,768, **b** = 32,768, **c** = 2) the efficiency is 90% (10% of the messages are lost). This is a significant improvement.

In addition, if each memory module is able to handle two references simultaneously, the impact of memory hot spots or other nonuniform traffic load will be significantly reduced. The advantage of more predictable behavior from this may by itself be worth some extra cost.

The intuitive reason for this effect is that while on average a single memory has only one reference, there is a significant probability that it will have more than one. This same principle applies in the switching network at switches and concentrators. This is why we provide two channels for every switch port and more concentrator output channels than are required by the average load.

Figure 7 shows **P{k}**, the probability that the concentrator in Figure 3 has **k** input messages. The average input load is four messages, but by providing six output channels (indicated by the dotted line), we are able to reduce the loss to about 3%. When six or fewer messages arrive, they are all serviced. When more than six arrive, six are serviced and the remainder are lost. So, for **k** = 7, only one message is lost. We could reduce the loss further by providing more output channels.

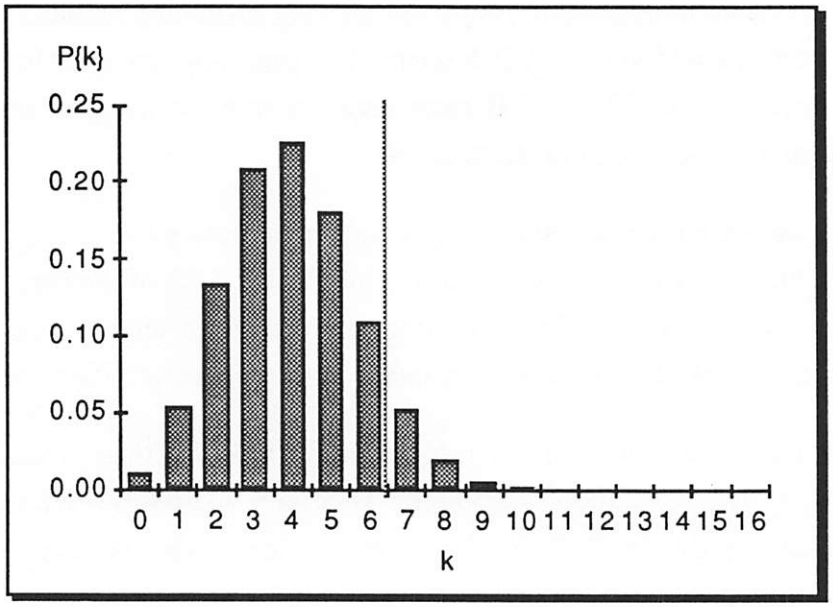

Figure 7 — P{k} For a 16 X 6 Concentrator

A Multistage Switching Network

The interconnection network of the Monarch is a cascade of interconnected switching elements of the form described above. We can repeatedly apply Equation 3 to each stage of the network to determine the network's overall performance. Table 2 gives the efficiencies for the 32-processor configuration in Figure 3.

Table 2 — Effect of Contention on 32-Processor Design

	a	b	c	Efficiency	Loss
Left Switch	4	8	2	98.5%	1.5%
Concentrator	16	1	6	97.3%	2.7%
Right Switch	6	4	2	93.3%	6.7%
Total				89.4%	10.6%

The 65,536-processor Monarch has seven stages of interconnect, including one in the memory controller. The configuration of each component and its efficiency are summarized in Table 3. Overall, the system has 16% contention.

Table 3 — Effect of Contention in a 65,536-Processor Monarch

	a	b	c	P_a	Efficiency	Loss
Switch 1	8	16	2	1.0	97.7%	2.3%
Concentrator 1	32	1	12	.24	99.3%	0.7%
Switch 2	12	16	2	.65	97.5%	2.5%
Concentrator 2	32	1	12	.24	99.5%	0.5%
Switch 3	12	8	3	.63	98.6%	1.4%
Switch 4	12	16	2	.62	97.7%	2.3%
Total Interconnect					90.7%	9.3%
Memory Controller	8	1	2	.11	93.2%	6.8%
System Total					84.5%	15.5%

The interconnection network in the Monarch is carefully engineered to achieve low levels of contention. It is also highly cost-effective since, in most configurations, there are about as many custom ICs in the network as there are processors in the machine — the network typically adds less than 25% to the cost of the machine. The network is highly flexible; it can support machines with only a few processors or machines with tens of thousands of processors.

2.2 Monarch Processor

In uniprocessor computer design, the speed of access to memory largely determines the performance of the machine. Only a modest amount of work can be done with data in the registers before more data must be drawn from memory. Fortunately, with one processor and one memory, it is often possible to locate them close together with short delays and wide data paths between them.

However, it is physically impossible to connect tens of thousands of processors *directly* to a large memory system — some form of communications system must intervene. Whatever form it takes, it introduces delay in getting to memory — delay that can slow down the processors.

One way to get around this problem is to place a small amount of memory next to each processor (if it is close, access can be fast), restrict the processor to accessing only this memory, and have the processors communicate in some other way. Message passing machines take this approach.

Another solution is to put a cache next to each processor. Presumably, most of the processor's memory accesses will be provided quickly by the cache, and the impact of delay in getting to remote memory will be diluted.

The Monarch design accepts the delay of the network. We have designed the fastest and best interconnection network that we could and then designed the processor and memory to match the network. The result is a processor with a peak performance of 6 MIPS or 2 MFLOPS (64-bit) — moderately fast but not outstanding by today's standards. However, it is a single chip 64-bit processor with on-chip floating point and MMU that requires only 3 DRAMs for support and interfaces directly to a high-speed switch. This CMOS design uses approximately 170,000 transistors, making it comparable in size to the Motorola 68020. The high processing power of the machine comes in the aggregation of thousands of these processors.

While the memory reference rate sets a limit to the performance of the machine, not all instructions require data from memory. In the Monarch processor, six ordinary (register to register) operations or two floating point operations, or a mixture, can take place while a memory reference is flowing through the network.

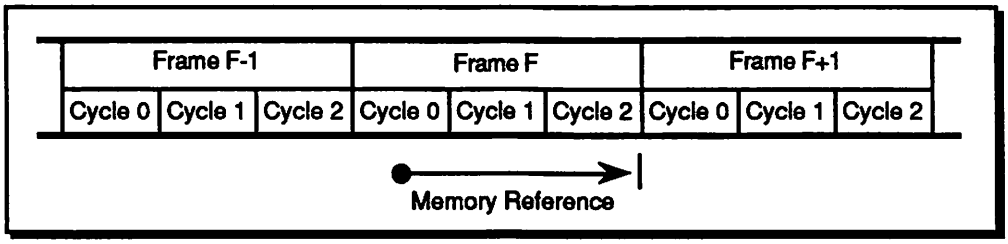

Figure 8 — Frame Timing

As Figure 8 shows, each frame is subdivided into three *cycles*. The processor can start a new instruction in each cycle, and most instructions finish in the same cycle. Memory reads take a whole frame. Thus, the data from a memory read that starts in cycle 0 of frame F is available to an instruction that executes in cycle 0 of frame F+1; of course, cycles 1 and 2 of frame F may be used for any instructions that do not depend on that memory data. Floating point operations also require more than one cycle to complete, and they also overlap with other instructions.

The processor has six independent functional units: two ALUs, a floating point multiply and divide unit, a floating point add and subtract unit, a memory interface unit, and an I/O unit. The instruction format packs two operations into one instruction, allowing the processor to perform two operations per cycle using two different functional units at the same time. This is the basis of the 6 MIPS peak instruction rate.

The Monarch processor has a *load-store* architecture like the Cray, CDC 6600, and the current generation of RISC processors. Memory reference instructions only load or store between memory and registers; no arithmetic or conditional operations deal with data in memory. This has allowed a straightforward extension; the Monarch is a load-store-*steal* architecture. *Steal* is the only mechanism for interprocessor communication and synchronization. This functions as a *load* instruction, but it changes a tag in the memory to mark the *stolen* word as unavailable. A processor attempting to load or steal the contents of a stolen location receives a reply that the location is stolen. That processor will retry the load over and over until another processor stores into that location. It will then receive the new contents of the location. This allows both the efficient construction of higher level synchronization mechanisms and a primitive form of "data flow" operation. Our studies strongly suggest that *steal* facilitates efficient implementation of many parallel programs.

In addition to 64 bits of data, every word in the machine has 8 tag bits. Two are used to aid garbage collection algorithms; the other six provide 64 possible tag codes. One code is used to indicate that a memory location has been stolen, one indicates that the data is

integer, one that it is floating point, and one indicates that the memory reference contains no data. The other 60 codes distinguish between pointers and other types of data and can be assigned by the software. These tags might be used to support generic arithmetic or diagnostic facilities (memory reference traps), for example. We have provided a generalized mechanism that we hope will be useful in the future.

While it might appear that we should purchase a commercial processor and interface it to the network (as we have done with the DRAMs in the memory modules and did with the Butterfly machine), the impediments to this approach are significant. High-end commercial microprocessors such as the Motorola 88000 are designed for a physically small system where the processor is the heart of the system. Memory is close to the processor, so the latency to memory is a fraction of a microsecond. Wires are cheap, so data paths are 32 bits wide.

In the Monarch, the memory is farther away, latency is almost a microsecond, wires are expensive so it is worthwhile to run a few wires at high speed, and the processor is duplicated many times. Parallel processing requires sophisticated interprocessor synchronization (in our case the *steal* instruction) — a test and set instruction or compare and swap is not good enough. Finally, a highly parallel shared-memory machine will be called on to solve large problems and requires a very large memory system (32-bit addresses are too small). As the design has progressed, it has become clear to us that a machine with these qualities could not be designed today using commercial microprocessors.

Caches

One of the system issues we considered is the use of cache memories to speed up accesses. Here it is important to distinguish two types of cache, instruction cache and data cache, because the two are accessed in very different ways.

Instruction Cache. The Monarch instruction memory acts like a software-controlled cache, with a capacity of 0.4 megabytes, allowing each processor to fetch instructions locally and reduce the demand on the switch and main memory. This is a sensible design because of the locality of instruction references; instruction caches may achieve a hit rate of nearly 99%.

Data Cache. In contrast, data caches rarely achieve hit rates of 90%, because of the "read once" nature of data in large databases, sparse matrices, hash tables, sequential records,

heaps, etc. Unfortunately, a cache miss often results in an order of magnitude longer access time than a cache hit, so a data cache with a hit rate of less than 90% throws away half the performance of the machine while imposing a significant cost. In addition, maintaining consistency of these caches by multiple processors is a complex and challenging problem. Furthermore, in a parallel processing system, shared data might migrate from cache to cache. Since caches add to system cost and complexity and frequently provide poorer access times in the case of a cache miss, it is very important that the cache hit rate be high just to break even. It is impossible to be confident that this cache hit rate will be acceptably high for arbitrary programs and data sets which have not yet been developed. Accordingly, we have taken the safe and conservative position of using processors which don't need data caches.

Fewer Faster Processors

One design alternative we considered and rejected was the use of fewer, faster processors. For example, instead of 65,000 six MIP processors, one might use 6,500 processors, each ten times faster (60 MIPS). To make this design comparable to the Monarch, thirteen of these processors must fit on a board (instead of 128) at a parts cost of about one thousand dollars for each processor instead of about one hundred dollars.

Simply scaling the machine with a processor that is ten times faster would require an interconnection network that is also ten times faster with access to 6500 memory ports in 100 nanoseconds. The imagined design is still quite large. The network would have ten times fewer ports. We can imagine increasing the bit rate by a factor of ten through GaAs, or using more paths in parallel, but it is hard to reduce the latency across the switch. With 20-foot cables in the network, speed of light delays to and from the memory are about 50 nanoseconds, leaving 50 nanoseconds for switching and memory access. This requirement is bound to increase the cost of the machine.

It would be nice to cut the size of the machine and reduce the cable length, but remember, we are planning 500 processor boards large enough to hold thirteen 60-MIP processors and their supporting logic. Much denser packaging may not be available. In any case, exotic techniques such as liquid cooling or wafer scale integration could be applied to either machine.

The standard solution would be to install a data cache with each processor to provide the data references locally. As noted above, we are not confident that the percentage of cache hits will be acceptably high in a parallel processing environment.

What have we accomplished by this change? To get the same performance as a 65,000-processor Monarch with these fast processors still requires a massively parallel architecture (6,500 processors). The step from one processor to many is difficult to deal with, but we don't believe the step from 6,500 to 65,000 is very significant. In our analysis of many representative problems, we found enough parallelism in the large problems.

In summary, we believe that the true issue in the trade-off between the speed of the processors and the number of processors is not one of cost/performance, but one of algorithm structure. In a design with data caches, the programmer must ensure the locality of data accesses. In a design without data caches, he must find and exploit a greater degree of algorithm parallelism. In either case, to achieve massive performance, massive parallelism will be required, even with very fast processors.

Our opinion, buttressed by several years of programming both machines with many processors and machines with local memory, is that it is *much easier* to deal with many processors than it is to deal with the restrictions imposed by a non-uniform memory structure.

2.3 Memory System

The Monarch has a single contiguous memory space to which all processors have equal access. This uniformity in the memory is key to the shared-memory architecture of the machine. With a capacity of 16 gigabytes and a bandwidth of 512 billion data bytes per second in its largest configuration, the memory system is able to meet the demands of 65,536 processors.

Throughout the Monarch, we use cost-effective components in quantity to achieve high performance. In the memory system as well, we use thousands of memory modules to supply the performance needed by thousands of processors. These memory modules are independent identical units consisting of one custom CMOS chip and two banks of memory, each with five 1-megabit DRAMs. After allowing for error correction and tag bits, a memory controller has a capacity of 512K bytes of data and can support two simultaneous memory references per frame per phase. The memory module is illustrated in Figure 9.

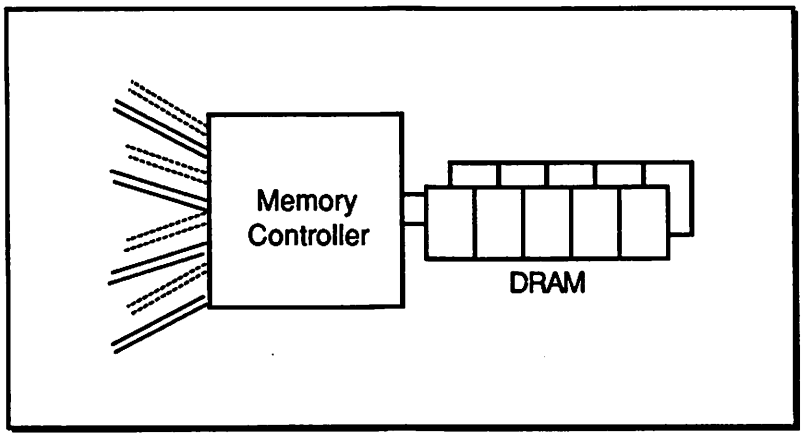

Figure 9 — Memory Module

The Monarch memory system is designed to accommodate the peak load generated when every processor makes one memory reference every frame. However, the dynamic memories used in the memory system are fast enough that they can perform two operations in the time of one frame. Thus, a Monarch needs only half as many memory modules as it has processors.

The memory controller is a custom CMOS chip being designed by BBN. It performs three memory functions, read, write, and steal, based on requests from the switch. The memory controller also provides error detection and correction on the DRAM contents. Memory refresh is synchronized across the machine so that its impact is limited.

The memory module is designed to supply memory bandwidth in the most cost-effective way. To implement the memory itself, we use 256K by 4 DRAM chips for the following reasons:

- We can achieve very high bandwidth (about 80 Mbit per second per chip) by using these devices in page mode.
- They provide a better value in bits per dollar than any other chip, because they are an international, high-volume commodity chip subject to intense price competition. Using them makes it economically feasible to create a "memory rich" machine with a huge amount of main memory. This means that a programmer can afford to make space/time trade-offs to achieve a significant speedup.[1]

[1] We continue to hold this belief in spite of the increase in megabit DRAM prices due to the worldwide shortage of 1988.

Each bank of memory consists of 5 DRAMs. Since each DRAM is 4 bits wide, the data path to the memory controller is 20 bits wide. When a memory reference occurs, four page mode operations are used to transfer 80 bits; 64 bits of data, 8 bits of tag, and 8 bits of error control.

The earlier analysis showed that if memory modules were only single ported and all the processors made random accesses every frame, the resulting memory contention rate would be about 37% (see Equation 4), which would cause a corresponding loss in total system throughput. We chose to dual port the memory controllers, so they could service two simultaneous independent requests. This reduced memory contention to about 10%. Actually, the memory system performance is somewhat better than this (about 5% contention) since contention in the switch reduces the load on the memories to about 90%.

We have dual ported the memory by providing two banks of DRAMs on each memory controller. This allows each controller to service two simultaneous memory read requests; one from each bank. Write requests cause the two memories to be written simultaneously so that the data is always the same in each.

Of course, the memory controller does read combining with its 8 inputs just as the switch does, so accesses to the same location are combined and only one read is performed.

In addition, the memory controller has a small queue of write requests. Thus the memory controller may service, say, two read requests and four write requests during the same frame. Write requests are serviced when time allows. Rough analysis indicates that this should reduce memory contention by another factor of two, assuming one-third of the data references are writes.

From the point of view of a Monarch processor, the memory system begins at its switch port. The network performs part of the memory operation, routing messages to the correct memory module and interleaving successive words of the address space across the memory modules so that word 0 goes to memory module 0, word 1 goes to memory module 1, and so forth. Interleaving the memory address space serves to disperse common linear data structures across the memory modules and, in doing so, to spread the switch traffic as well. With as many as 32K memory modules, even rather large data structures are dispersed.

In summary, the Monarch memory design achieves very high performance (64 billion references to shared data per second with less than 5% memory contention), with large capacity (17 gigabytes of shared data).

2.4 Maintenance and Diagnostic System

Control of the Monarch hardware is accomplished by a Maintenance and Diagnostic System (MDS) to which every custom chip in the machine is attached. This allows a separate maintenance computer to control and monitor the machine independently of the operational software. The functions of the MDS fall into three groups:

1. Initial system configuration and loading, starting and stopping.
2. Fault monitoring and recovery.
3. Fault diagnosis.

The custom devices are designed so that they can be adapted to a particular machine configuration. Values such as the length of the frame, the expected arrival time of messages, and many others can be set according to the machine size and the location of the chip in the machine. The MDS can also access the memory in the machine to load bootstrap programs. Even the synchronization of frame time and the establishment of data recovery across the machine are controlled by the MDS.

The initial configuration of the machine may include information about faults in the hardware (such as loose connections or bad devices) or components that are out of service. The MDS can set state bits in custom chips to disable any high-speed link in the network. This provides a mechanism for bypassing faulty chips in the interconnection network by disabling all links connected to the faulty chip. It can also control the configurations of switch chips and disable memory banks. These facilities allow the MDS to flexibly configure the machine even in the presence of numerous faults.

During normal system operation, error detection circuitry monitors every message flowing through the interconnection network, every instruction memory and global memory reference, and clock and signal timing throughout the machine. In case of a fault, the chip takes whatever action is appropriate at the time and sets a status bit that the MDS can read as it scans the chips periodically. In the case of correctable errors, such as soft memory errors, the MDS may accumulate statistics or excise the defective component for more thorough off-line diagnosis.

The MDS is responsible for diagnosing faults in the machine. Special facilities are built into the custom devices to allow this. For example, every custom chip is able to control and monitor its input and output pins. Since all of the signalling wires in the machine terminate in our custom chips, this facility allows the MDS to perform a full continuity check on the data paths of the machine.

Through these fault discovery and recovery mechanisms, we may to be able to recover from many faults without bringing the machine down. Even when this is not possible because of the impact of a fault, we should be able to restart and reconfigure the machine quickly.

2.5 Packaging

Monarch machines vary in size from 1,024 to 65,536 processors and from 568 megabytes to 17 gigabytes of memory. All of these machines can be built out of the same key components:

- A processor board with 128 processors and a 16-way switch.
- A memory board with 128 memory controllers and a 384-input switch.
- A switch board with 192 input ports and 16 output ports with 12 channels each.
- Utility boards for clock and MDS distribution.

Because of the dynamic delay adjustment technique, a Monarch can be physically small or physically large without too much concern for signal or clock skew. The size of the machine is limited not by clock skew, but rather by the losses in the cables that interconnect cabinets.

At the low end, we can imagine a machine with eight processor boards, four memory boards, one utility board, and no switch boards. This machine would have 1024 processors and 262 megabytes of memory. It would fit in a single card cage about 18 inches wide, 24 inches high, and 36 inches deep. It would dissipate about 10 kilowatts and could be cooled by forced air.

The largest machine uses the same boards, but requires special cabinetry. An initial design for a 65,536-processor machine fills much of a 40-by-40 foot room, as shown in Figure 10.

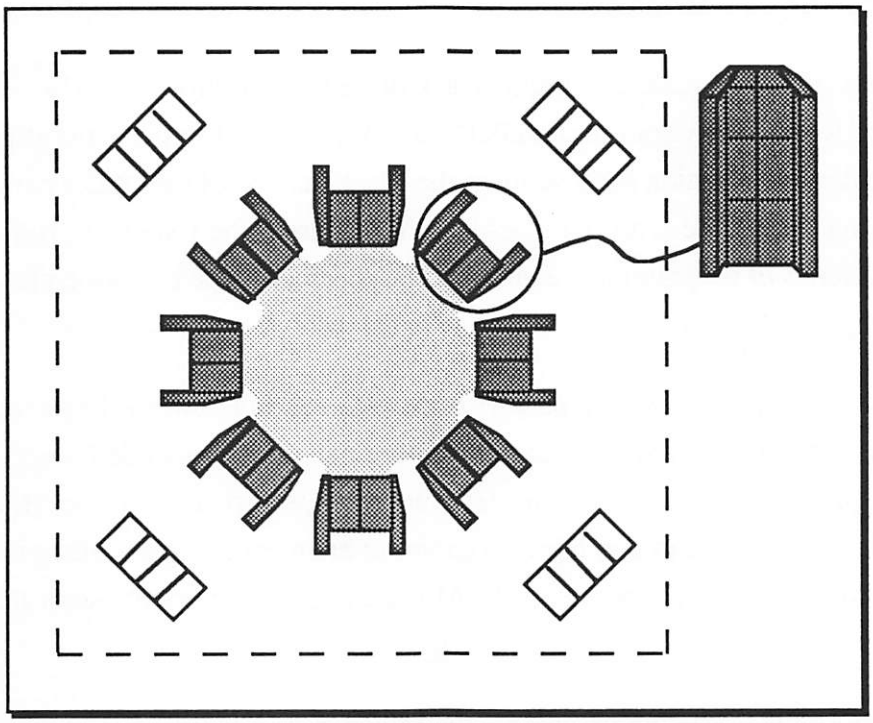

Figure 10 — 65,536-Processor Monarch Floorplan

The central machine occupies 16 cabinets arranged in a circle. Inside the circle is a wiring mat which interconnects the cabinets. The other cabinets arranged around the room contain disk drives, AC power and cooling control and distribution cabinets, and the MDS maintenance computers. This design is quite dense and requires water-cooled intercoolers within the cabinets to remove the heat generated by the computer. The reader should take this packaging example lightly since as the design continues, improvements in the packaging design are to be expected.

3. Software for the Monarch

Programmers of a new parallel computer ask three key questions: 1) How do I use the system?, 2) How do I program in parallel?, and 3) How well will my program run? We will answer the first question by describing the Mach version of the UNIX operating system that we intend to provide with the machine. We address the issues of programming by presenting a series of programming examples that illustrate some key principles involved in programming the Monarch.

While it is difficult to assess the performance of a machine until it has executed many benchmarks and many real application programs, we have coded over two dozen benchmark programs in an attempt to determine the performance of the Monarch. The results of this study confirm that large programs contain enough parallelism to use 65,536 processors and that the architecture of the Monarch permits efficient use of the machine's resources.

At the inception of the Monarch design, we selected the shared-memory architecture in order to present a very powerful and simple machine model to the programmer to make the programming task easier. We spent our effort building a high-performance interconnection network between the processors and memory so that the programmer could ignore the structure of memory. We also avoided adding special hardware mechanisms to improve the special cases that always come up during a design. It is hard enough to program a uniprocessor, and we didn't want to add to the difficulty any more than was necessary to gain the benefits of parallel processing. This in itself is unusual — most massively parallel machines demand major changes in programming style.

3.1 Mach

In order to present a familiar software environment, the Monarch will use the Mach variant of the UNIX operating system. Mach is a new operating system developed at Carnegie-Mellon University (CMU) with the support of the Defense Advanced Research Projects Agency (DARPA). It has been designed to provide a standard for extending the UNIX environment into the multiprocessor arena. Mach provides complete Berkeley 4.3BSD UNIX compatibility while significantly extending the UNIX notions of virtual memory management, processor management, and interprocess communication.

Mach takes an object-oriented approach to operating system services. It provides five fundamental objects visible to the user: a communications *port*, a collection of data objects called a *message*, a *virtual memory object*, an execution environment called a *task*, and *threads* that function as virtual processors. These five basic objects were described in Rashid, R.F., "Threads of a New System," *UNIX Review*, August, 1986 as follows:

- "A *task* is an execution environment in which threads may run. As the basic unit of resource allocation, a task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory)." A task address space consists of an ordered collection of mappings to memory objects (see below). "The UNIX notion of a *process* is thus represented in Mach by a task that has a single thread of control."
- "A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. Mach differs from UNIX in that multiple threads may execute simultaneously in a common address space. All threads within a task share access to all task resources." In particular, all share one virtual address space.
- "A *port* is a communication channel—a logical queue for messages protected by the kernel. As the reference objects of the Mach design, ports are used in much the same way that object references are in an object-oriented system. *Send* and *receive* are the fundamental primitive operations on ports."
- "A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports."
- A *memory object* is a collection of data provided and managed by a server. It can be mapped into the address space of a task.

UNIX popularized the concept of a process as a virtual processor. Mach extended that notion to parallel processing. In Mach, a process (a set of code and an execution environment) is called a task. A single task may be executed by one or more threads. Each thread represents a virtual processor with its own set of registers, program counter, and stack pointer. This virtualization of the parallel processor provides a machine-independent environment in which to develop parallel programs.

In Mach, all of the threads running in a task share the same code and data space, as shown in Figure 11. If two such threads make a reference to virtual memory using the same

address, they will reference exactly the same physical memory location. Even a thread's stack is independent only if other threads refrain from interfering with it.

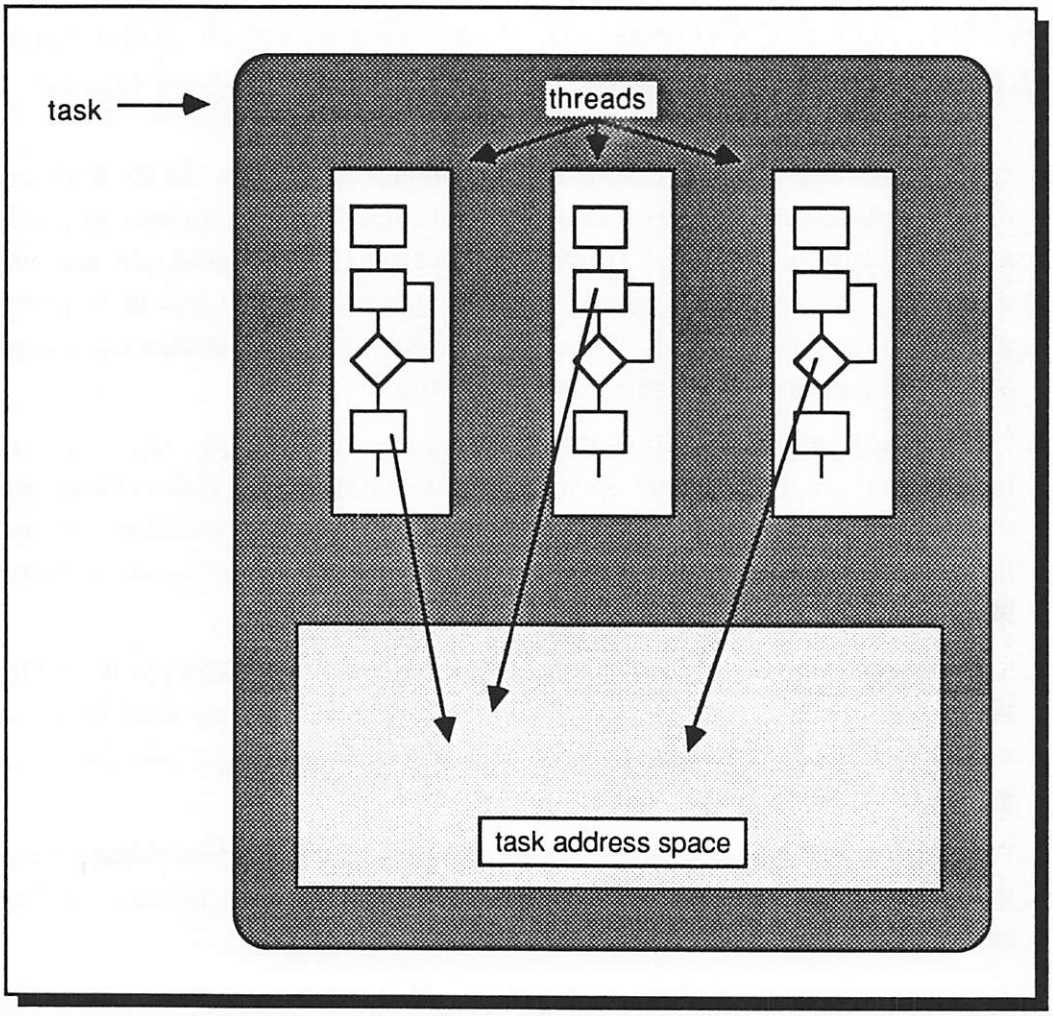

Figure 11 — Tasks and Threads

For us, this is an ideal parallel programming model. It allows the programmer to write a single program, use pointers and shared-memory communications, but run an arbitrary number of processors (threads) over the code. The allocation of all system resources is dynamic and simple. Mach threads are a light-weight form of process in contrast to UNIX's heavy-weight processes. Spreading users across a collection of processors is an interesting *space-sharing* variant of traditional time-sharing, in which a single processor's time is shared by the users of the system.

Mach makes few assumptions about the underlying hardware architecture. To date, Mach has been ported to such diverse architectures as uniprocessor and multiprocessor VAX™

systems, the IBM®RT and Sun Workstations®, the Encore Multimax multiprocessor, the Sequent Balance 21000 multiprocessor, and the BBN Butterfly™ GP1000 parallel processor.

3.2 Parallelism for UNIX Commands

Parallelism exists at many levels in a programming system. The programming examples in a later section illustrate parallelism at the loop and algorithmic levels, but it is often present at the level of shell commands as well. UNIX already supports the notion of performing operations in parallel from the shell. For example, an ampersand at the end of a shell command indicates that the command should run in the background, leaving the shell free to execute new commands.

Another form of parallelism in UNIX is the conceptual pipelining of processes. For example, the shell command

```
tbl | eqn | troff | print
```

tells UNIX to feed the output of the `tbl` program into the `eqn` program, the output of `eqn` into `troff`, and the output of `troff` into the `print` program. UNIX assumes that the segments of the pipeline can be run in parallel with later segments operating on data as it becomes available. In the Monarch shell, the segments could run on separate processors.

The UNIX shell also contains a for-loop command. A parallel version, `pfor`, could perform all its iterations in parallel using multiple processors.

From a program developer's point of view, one of the more useful ways to exploit parallelism is with a parallel version of the UNIX facility `make`. This UNIX command aids in maintaining the set of files involved in a large programming project. Given a description of the interdependencies (a makefile) between these files, the `make` command will examine the time each file was last modified and recompile those object files which have become out-of-date with respect to their sources. A parallel version of `make` will allow many of those recompilations to execute in parallel.

These examples of parallel operation at the shell level illustrate one benefit of doing development on the parallel machine itself. Further improvements may be possible where other UNIX utility programs can be decomposed and parallel processing applied. However, we expect that most of the parallel programming focus will be on writing new programs

and modifying old ones to take advantage of algorithmic parallelism. For this, appropriate languages are needed.

3.3 Language Support

While better languages for parallel processing are needed (indeed better languages are also needed for uniprocessors), we have developed our plans around the standard languages of today, C and Fortran, so that today's programmers can feel comfortable with the machine.

The Monarch C compiler will provide the same functionality as the Berkeley UNIX 4.3 C language, including interface, outputs, and libraries. Thus, application programs and utilities developed under Berkeley UNIX or compatible operating systems will be able to be compiled and executed in the Monarch environment.

The compiler has been extended in several ways to support parallel processing. Two new data types, *shared* and *private,* have been introduced to deal with the memory architecture. If storage is declared to be *shared,* one copy is allocated in memory and all of the threads access it by the same name. If storage is declared to be *private,* memory will be allocated so that each thread has its own copy of the data structure and each thread will access its own copy.

In order to support the *steal* operation, the compiler recognizes the pseudo-function `steal()` and converts it into the *steal* memory reference. The argument to `steal()` is the address of a single word. The previous contents of the word are returned by the function. These extensions alone allow parallel programming on the Monarch. However, we also provide a library of convenient functions for parallel programming.

In order to support the shared-memory model, the Fortran notion of data storage must be modified. Common blocks may have an associated attribute of *shared* or *private.* A common block specified as `shared common` is a single block shared by all threads in a task. When each thread has a private copy of a common block it is specified as a `private common` block.

A basic unit of parallelism is a parallel do loop invoked by the command `do parallel` which converts the immediately following do loop into a parallel loop.

3.4 Parallel Programming Examples

Rather than trying to describe the general principles of programming the Monarch and how it feels to program it, we have elected to work through a series of examples using the programming language *C*. In order to understand these examples, the programmer needs to know some more about the environment in which they run.

By the time an application program gains control of the machine, many events have already occurred. Certain things happen when the machine is powered up, others when a user logs on and begins to access the machine, and still others when the user asks to have a program run. Collectively, these events are the province of the operating system, and they are invoked by the runtime executive, which reserves certain regions of memory for specific purposes and ensures that the user's program, the library routines it needs, and the operating system kernel are all in their proper places.

Other regions of memory are set aside for data by a memory allocator that services user and loader requests for blocks of storage. The memory allocator resembles its uniprocessor counterpart, supplying blocks of contiguous locations with no attempt to align the blocks in any particular way, except to prevent wasted storage. Depending on the programming language used, main memory is allocated at load time (e.g., for Fortran programs) or during an initialization phase (for languages such as C).

Because the Monarch has no concept of memory locality, memory allocation in no way depends on the application program. Architectures that make use of locality face a much more complex memory allocation problem because the structure of an application program influences the placement of its data in memory.

When an application program gains control, the machine is in a prescribed state. A different region of memory is allocated for each thread's stack, and a register in each thread is assigned to serve as a stack pointer. Another register contains a number that serves to distinguish one thread from another. In addition, all processors have access to a shared data structure that contains information such as the number of threads in this task, the actual number of processors running the task, and the number of processors available in the machine.

The number of threads in a task is specified when the task is specified. Either the programmer will specify the number of threads or the operating system will choose. While the program is executing, it will be possible to fork additional threads or to stop a thread.

In these examples, we will keep the number of threads constant. When the program begins, all of the threads execute the subroutine `main()`, which contains or calls the code in the examples.

Two variables, `myproc` and `nprocs`, are predefined. `nprocs` is the number of threads in this task. `myproc` is a number between 0 and `nprocs-1` that serves to identify each thread in a task. `nprocs` is shared and `myproc` is private.

The function `barrier()` returns simultaneously in all threads only after every thread in the task has called it. It executes in $10 + \log_2 P$ frames (a frame is 1 microsecond). [Note: all logarithms used here are base 2.]

Given these ground rules, we can try a few simple examples: adding a million numbers and sorting a million numbers. For each example, we will provide a clear statement of the problem, the uniprocessor code to solve it, the parallel processor code, an analysis of the expected performance, and a commentary on the example. In the formulas that follow, we will use **P** to indicate the number of threads running in the task (`nprocs`) and **N** to represent the number of data items (one million in these examples).

3.4.1 Example 1 - Sum One Million Numbers

Given an array of one million real numbers (`Array[1000000]`), compute the sum of all the numbers in the array. Return the answer. The uniprocessor code is given in Figure 12.

```
float Sum(){
     extern float Array[];
     int i;
     float Total = 0.0;
     for (i = 0; i < 1000000; i++)
          Total += Array[i];
     return(Total);
}
```

Figure 12 — Summing 1 000 000 Numbers (Uniprocessor Example)

The parallel code shown in Figure 13 requires the introduction of a new private variable, `Subtotal`, to accumulate the partial results for each thread. Two barriers are required. The first assures that `Total` will not be used before it has been cleared. The second assures that the final answer will not be used before all of the processors have contributed their subtotals. Each processor returns the result of the calculation.

```
float Sum(){
     extern shared float Total, Array[];
     int i;
     float Subtotal = 0.0;
     for (i = myproc; i < 1000000; i += nprocs)
          Subtotal += Array[i];
     if(myproc == 0) Total = 0.0;
     barrier();
     Total = steal(&Total) + Subtotal;
     barrier();
     return(Total);
}
```

Figure 13 — Summing 1 000 000 Numbers (Parallel Processing Example)

In the uniprocessor example, we indexed from 0 to 999999. In the parallel programming example, we made a different choice. As written, the processors work through the array starting with their own thread number in steps of nprocs. This keeps the end test simple. A different way to do this would be to divide the array by the number of threads and assign a contiguous piece of the array to each thread.

After each subtotal is computed, the overall total can be computed. Processor zero initializes the total while all processors wait at the barrier. Then, each processor steals the total, adds in its subtotal, and stores the new total. There is no need for an explicit lock since the steal operation serializes the operations on the total.

The performance of the parallel routine is:

$$\text{Parallel Performance} = 21 + 2P + 2\log P + \frac{N}{P} \qquad \text{(Equation 6)}$$

How efficient is the machine in this example? We define the efficiency of a parallel processor as the ratio of: 1) the time it would take to solve the program using a uniprocessor running the uniprocessor code to 2) **P** times the time it would take to solve the problem using a parallel processor having **P** processors using the parallel processor code. Inefficiency is caused when the parallel algorithm is less efficient than the uniprocessor algorithm or the parallel machine is less efficient than the uniprocessor executing these algorithms.[1]

[1] While this definition of efficiency is useful for our analysis, many actual users are more concerned with the total time to solve the problem, the total size of the problem solvable, or the effective cost to solve the

A uniprocessor running the uniprocessor code would solve the problem in time:

$$\text{Uniprocessor Performance} = N \qquad \text{(Equation 7)}$$

The parallel processor efficiency is:

$$\text{Efficiency} = \frac{N}{N+P(21+2P+2\log P)} \qquad \text{(Equation 8)}$$

$$\approx \frac{1}{1+\frac{2P^2}{N}} \quad \text{[within 2\% if N>500000]} \qquad \text{(Equation 9)}$$

The machine is 50% efficient when $P=\sqrt{N/2}$; in this example, when there are 700 processors. If we had only 200 processors, the algorithm would be 90% efficient, but with 65,536 processors, the efficiency would only be 0.01%.

Clearly, in a machine with many processors, a better approach is needed. This code simply adds each subtotal to the total one at a time, requiring **2P** references to the total. A more sophisticated algorithm would use a binary tree to combine these subtotals in **O**(log **P**) time. The Monarch parallel subroutine library contains a number of common logarithmic-time operations (`barrier` is one of them). The routine `LogSum(subtotal)` combines subtotals from each thread and returns the answer. All processors return from LogSum simultaneously. LogSum requires $11 + 2 \log P$ frames. We will come back to the principles behind logarithmic operations later.

```
float Sum(){
     extern shared float Array[];
     int i;
     float Subtotal = 0.0;
     for (i = myproc; i < 1000000; i += nprocs)
          Subtotal += Array[i];
     return(LogSum(Subtotal));
}
```

Figure 14 — Summing 1 000 000 Numbers (Improved Parallel Example)

The execution time of the improved routine is:

$$\text{Improved Performance} = 11 + 2 \log P + \frac{N}{P} \qquad \text{(Equation 10)}$$

problem. A parallel processor need not be optimally efficient — just efficient enough in the dimension the user cares about.

The efficiency is:

$$\text{Improved Efficiency} = \frac{1}{1+(11+2\log P)\frac{P}{N}} \qquad \text{(Equation 11)}$$

In the improved code, the efficiency is 50% if 24,000 processors are used to add one million numbers. If only 1000 processors are used, the efficiency is 97%, and if 65,536 are used, the efficiency is 26% and the problem is solved in 59 microseconds.

Now that we understand the basics of Monarch programming, let's look at a problem that shows off the advantage of a shared-memory machine.

3.4.2 Example 2 - Sort One Million Numbers

Given an array of integers (`Data[DataCount]`), place the numbers in numerical order in a second array (`Output[DataCount]`).

```
BucketCount = 4* DataCount
int bucket[BucketCount], Data[DataCount];
FLAG = 1<<63;

sort(){
     Initialize_Buckets();
     Insert();
     Compact();
}

Initialize_Buckets(){
     int i;
     for (i = 0; i < BucketCount; i ++)
          bucket[i] = FLAG;
}

Insert(){
     int i, key, old, x;
     for (i = 0; i < DataCount; i ++){
          x = Data[i];
          for (key = x >> 42;; key ++){
               old = bucket[key];
               if (old == FLAG){
                    bucket[key] = x;
                    break;
               }
               if (old > x){
                    bucket[key] = x;
                    x = old;
               }
          }
     }
}

Compact(){
     int i, j, x;
     j = 0;
     for (i = 0; i < BucketCount; i++){
          x = bucket[i];
          if(x != FLAG) Output[j++] = x;
     }
}
```

Figure 15 — Uniprocessor Sort

Sorting on a uniprocessor is not exactly straightforward. The programmer must first decide what sorting technique is appropriate for the number of input data elements, the statistical distribution of the input data, the memory configuration of the machine, and the

time available to write the program. Techniques exist that require time proportional to NlogN, or N^2, etc. In many cases, there is a significant difference between the average execution time and the worst-case execution time. See [Knuth 73] for a thorough examination of sorting techniques.

We have chosen a sort that uses a lot of memory because the Monarch has a lot of memory and we are willing to assume that the numbers are uniformly distributed between -2^{63} and $2^{63}-1$. To keep the code simple, we have ignored the case where data is equal to the flag.

The algorithm has three phases. First, the buckets are all marked empty by placing a flag in each bucket. Then, each item is inserted into the array. The insertion point is determined by selecting only the 22 high-order bits of the data; thus, 2^{22} (about 4 million) buckets are used. If the bucket is empty, the data is inserted in the table. However, if the bucket is occupied, the new data is sorted into this and succeeding buckets until an empty bucket is found. Finally, the buckets are scanned in order and the non-empty buckets are dumped to the output.

The performance of this uniprocessor algorithm is as follows:

$$\text{Uniprocessor Performance} = (4+2R+2k)\,N \qquad \text{(Equation 12)}$$

Where **N** is the number of data elements (DataCount), **R** is the ratio of the number of buckets to the number of data elements, and **k** is the expected likelihood that a bucket is already full when needed. **k** is approximately equal to half the occupancy of the bucket array (1/2**R**). We will choose an **R** of 4 for our examples and ignore the effect of **k** in the rest of our calculations. Then, the uniprocessor performance is 12 **N**.

```
BucketCount = 4 * DataCount;
shared int bucket[BucketCount], Data[DataCount];
FLAG = 1<<63;

sort(){
     Initialize_Buckets();
     barrier();
     Insert();
     barrier();
     Compact();
     barrier();
}

Initialize_Buckets(){
     int i;
     for (i = myproc; i < BucketCount; i += nprocs)
          bucket[i] = FLAG;
}

Insert(){
     int i, key, old, x;
     for (i = myproc; i < DataCount; i += nprocs){
          x = Data[i];
          for (key = x >> 42;; key ++){
               old = STEAL(bucket[key]);
               if (old == FLAG){
                    bucket[key] = x;
                    break;
               }
               if (old > x){
                    bucket[key] = x;
                    x = old;
               }
               else  bucket[key] = old;
          }
     }
}

Compact(){
     int i, j, count, BlockSize, x;
     count = 0;
     BlockSize = BucketCount/nprocs;
     for (i = myproc*BlockSize; i < (myproc+1)*BlockSize; i++)
          if(bucket[i] != FLAG) count++;
     j = LogCumulativeSum(count);
     for (i = myproc*BlockSize; i < (myproc+1)*BlockSize; i++){
          x = bucket[i];
          if(x != FLAG) Output[j++] = x;
     }
}
```

Figure 16 — Parallel Sort

For a parallel version of this algorithm, we found parallel implementations of all three phases and placed synchronization barriers between them to ensure that all of the processors have completed one phase before any move on to the next one.

As with the previous example, we can divide initialization between the processors quite easily.

We can also divide up the input data stream and let each processor insert its fraction of the data. The only obvious difference between this code and the uniprocessor version is that when we reference the bucket table, we steal the data. Doing this ensures that no other processor can access this bucket until we have finished with it. It does require writing back the old value into the bucket when the old value is less than x. The table of buckets stays in order no matter which processor wins each steal. However, this lock could be a point of contention when two processors access the same bucket. This happens with a frequency of 65000÷4000000 or 1.6% of the time — we ignore this effect.

The third phase, compacting the output list, is done by dividing the array of buckets into ranges and assigning each processor a range of buckets. Each processor counts the number of items in its buckets and then stores the items in the output array starting in the correct place. The correct place depends on how many items were found by those processors with a smaller virtual processor number. Processor 0 starts at zero; processor 1 needs to know the count from processor zero; processor 2 needs the sum of the counts from 0 and 1; processor 3, the counts from 0, 1, and 2; and so on.

This information can be computed using a parallel prefix sum (LogCumulativeSum) in time proportional to log P. LogCumulativeSum accepts the counts from each processor and returns to each processor the cumulative sum of the arguments from all processors numbered less than it. It runs in 17 + 3 log P frames.

In summary, the performance of this algorithm is:

$$\text{Parallel Performance} = 47 + 6 \log P + 15 \frac{N}{P} \qquad \text{(Equation 13)}$$

The parallel processing efficiency is:

$$\text{Efficiency} = \frac{12}{15 + \frac{P}{N}(47 + 6 \log P)} \qquad \text{(Equation 14)}$$

We can see that the parallel version requires one more scan through the buckets and three barriers more than the uniprocessor algorithm, but there are no serial bottlenecks. The change in the algorithm itself limits the efficiency to 80%. 65,536 processors running the code would be about 50% efficient.

This result suggests that given a large enough machine, the problem can be solved in 1/30,000 th of the time it takes on a uniprocessor. If the data set is larger, the speedup can be greater.

We have found that in these examples, it is important to avoid operations that require time proportional to the number of processors, and have invoked logarithmic operations instead.

3.5 Logarithmic Operations

When many processors team up to execute a parallel program, it is often necessary to combine the intermediate results generated by the individual processors into one composite result. Serial combining, where each processor in turn integrates its contribution into the final result, is inefficient because it takes **O(P)** time for **P** processors to combine their results. Logarithmic combining is a software technique for integrating **P** intermediate results into one composite result in **O(logP)** time.

Concern over processor synchronization has been a centerpiece of parallel computer theory for many years. Hardware facilities that aid interprocessor communications are frequently incorporated in the designs for parallel machines. These facilities have included the *test and set* instruction, the *compare and swap* instruction, the *full* and *empty* bits on memory, *fetch and op* (*fetch and add*), and synchronization busses. For a comparison of these mechanisms, see [Dubois 88]. The Monarch uses the *steal* operation along with a full/empty tag on every memory word as the hardware-level indivisible operation and *read combining* as an adjunct that allows read memory operations to be done in a fixed amount of time even with a very large number of processors. We have applied software to the task of implementing the mechanisms that synchronize algorithms.

We saw in the first programming example that the change from linear combining of subtotals to logarithmic combining of subtotals increased the size of machine that could achieve efficient operation. Perhaps we have not gone far enough. If we had hardware to perform the barrier and LogSum operations in one frame time, 64,000 processors would be able to achieve better than 50% efficiency on the first example. We are paying a penalty for doing the work in software. While it is hard to determine the break-even point where the additional cost of the hardware equals the efficiency lost by the software, our view is that the hardware to implement these operations in a flexible way is simply too expensive [Gottleib 87]. In contrast, software is quite flexible, allowing much more complex operations.

When one considers special hardware for this type of operation, remember that not all of the processors may be running the same task, so a wire bus that synchronizes all of the processors will not work. Also, if we are going to do fetch-and-op types of operations, then we will need both integer and floating point versions.

We used three logarithmic operations to solve the two sample problems: barrier, LogSum, and LogCumulativeSum. Examining how these functions are implemented will reveal more about the operation of the machine.

3.5.1 Barrier Synchronization

A barrier function is used to assure that all of the processors are finished with one phase of a computation before any proceed to the next phase. We can implement a barrier that executes in 10 + log **P** frames. The implementation of barrier requires a shared array (`Array[nprocs]`) that has been initialized so that all of its words are stolen. The algorithm leaves the array in the same state (except for element zero which is treated specially). This algorithm works for any number of processors, not just a power of two. `t_procs` is initialized to $2^{\lceil \log nprocs \rceil - 1}$; `phase` is a number that is different (e.g., incremented) each time `barrier` is called.

```
Barrier(phase)
int phase;
{
     int x;
     delta = t_procs;
     while (myproc < delta){
          if (myproc + delta < nprocs)
               x = steal(&Array[myproc+delta]);
          delta = delta/2;
     }
     Array[myproc] = phase;
     while(loadlp(&Array[0]) <> phase);
}
```

Figure 17 — Barrier Synchronization Subroutine

Barrier synchronization is a multistage operation with a total of log **P** stages, as illustrated in Figure 18. At each stage, the active processors are divided into two groups according to their processor numbers (`myproc`). Processors in the higher numbered half store into their element of the array and drop out of the computation by waiting at element zero. Processors numbered in the lower half wait for the corresponding higher numbered

processor by reading (and stealing) that processor's array element. The loop repeats, dividing the previous readers in half, until processor zero stores in element zero.

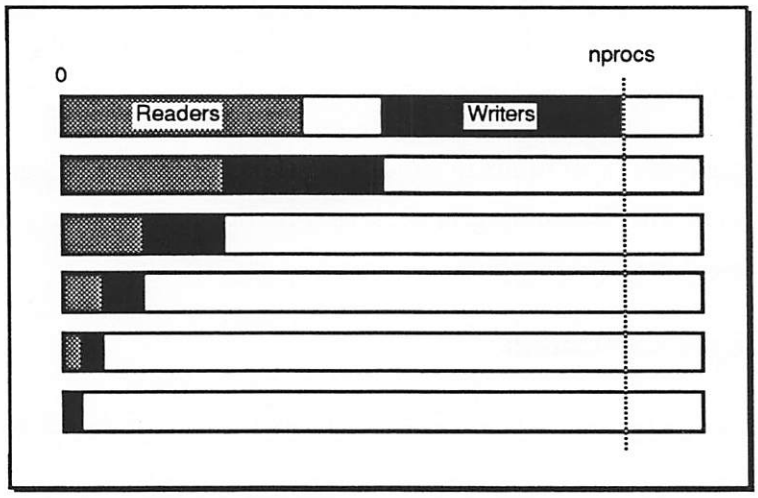

Figure 18 — Barrier Operation

The processors that drop out of the computation wait by reading element zero of the array. Of course all of the reads combine in the switch so one successful reference is seen by all. The processors wait until the memory location is filled with the correct value before moving on to the next phase of the algorithm. The waiting processors use a low priority read (`loadlp`) to decrease the load on the switch and allow processor zero to store.

In this code, there are three special effects. First, we have carefully chosen `t_procs` to be the greatest power of two that is less than the number of processors. The first time through the *while* loop, there may not be as many writers as there are readers, so some of the readers do not read the array at all. Second, in order to restore the array to the all stolen state, the readers *steal* instead of reading. Third, barrier will be called many times during the execution of a program. We must allow processors to proceed past one barrier while stopping them at the next barrier. This is accomplished by numbering each barrier with a different value of the argument `phase`.

3.5.2 LogSum

The function `LogSum()` combines subtotals from all of the processors into a total in time proportional to log**P**. It returns the total to all of the processors and acts as a barrier. LogSum uses the same prestolen array used by barrier plus an additional variable `phase_number` for synchronization. Its code is presented in Figure 19.

```
float LogSum(Subtotal, phase)
float Subtotal;
int phase;
{
     delta = t_procs;
     while (myproc < delta){
          if (myproc + delta < nprocs)
               Subtotal += steal(&Array[myproc+delta]);
          delta = delta/2;
     }
     Array[myproc] = Subtotal;
     if (myproc == 0) phase_number = phase;
     while (loadlp(&phase_number) != phase);
     return(Array[0]);
}
```

Figure 19 — LogSum Subroutine

The only difference between this code and the code for barrier is that while barrier stores and reads dummy data, LogSum stores subtotals and the steal loop accumulates subtotals. This code works for any number of processors and leaves the array stolen just as in the barrier code.

The running time of the code presented here is 11 + 2 log **P** frames.

3.5.3 LogCumulativeSum

Example problem 2 (Sort) illustrates a common application of a *parallel prefix operation.* Problem 2 compresses a sparse hash table to generate its final output. Each processor must know where its data belongs in the output array. Logarithmic cumulative sum computes this result in only 17 + 3log**P** frames.

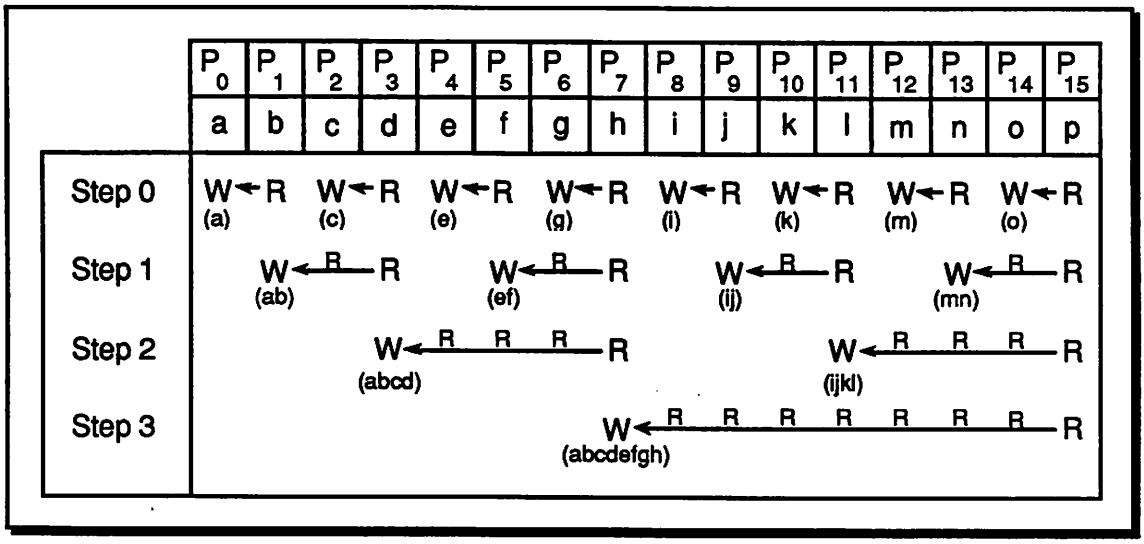

Figure 20 — Operations in LogCumulativeSum

Figure 20 shows the pattern of references generated by a set of sixteen processors participating in the computation of a running sum. There are five steps in the computation. At each step, processors marked with a *W* write their partial result into their element of the synchronization array. Processors marked with an *R* read from the location indicated by the arrow. In many cases, more than one processor will read the same location. We have labeled the input arguments *a* through *p* for processors 0 through 15. The partial result that each processor writes is shown in parentheses in the figure (we have omitted the operator between the letters, so *ijkl* means *i+j+k+l*).

For example, processor 10 starts out with *k*. At step 0, it writes *k* into element 10 of the array. In step 1, it reads the partial result *ij* from element 9 of the array (waiting for processor 9 to write if necessary). In step 2, processor 10 has nothing to do. In step 3, it reads the partial result *abcdefgh* from element 7 of the array, combines this with *ij* into the result *abcdefghij*, and is done.

Since more than one processor must read the same location, we cannot use *steal* instead of *read* to resteal the array. Before the computation, each processor steals its element of the array and synchronizes at a barrier before performing the sum. This initialization is included in the run time.

3.6 Recurring Themes

During the design of the Monarch, we have examined and coded several dozen examples. Several common themes emerge for programming this machine.

A machine with tens of thousands of processors must be operating in parallel all of the time. If it seems that part of the program cannot run in parallel, it should be reexamined. If part of the program requires each processor to do something one-at-a-time, this is a big problem, and a form of the algorithm that operates in logarithmic time should be sought. In the examples we have examined, we have found enough parallelism in almost every case.

The steal operation and read combining are powerful primitives for parallel algorithms. Frequently, an algorithm can steal a piece of data, thereby implementing mutual exclusion in a natural way. Since every word in memory has a full/empty tag, data structure locking can be done at the level of each word. A global lock is rarely needed. In this way, the machine provides a poor man's form of data flow.

It appears that we have succeeded in designing a truly shared-memory machine. The programmer does not need to think about where data is in memory. He may need to consider the use of registers versus memory, but there is little advantage in picking one memory location over another.

Because of read combining, it is easy to share values in the machine. Read combining acts like a broadcast mechanism. It also removes worries about sharing constants or global parameters.

Stragglers

If all processors simultaneously begin a computation that ends at a barrier, they may all reach the barrier at the same time. However, some processors may have more work to do in their part of the computation. If this effect is significant, then some approach to dynamic allocation of the computation will be needed. Contention can also cause some processors to arrive at the barrier after the others. We call these processors *stragglers*. In the analysis above, we have ignored stragglers.

In a 65,536-processor machine, with an average contention rate of 10%, we should expect that 6554 processors will have a collision on a particular memory reference. Of these 6554 processors, 655 will have another collision, 65 will have a third collision, 6 will have a

fourth, and 1 might have a fifth collision. In other words, if the calculation required one memory reference, most of the processors would finish in one frame and wait for the last processor to finish five frames later. (This ignores the change in load on the switch when the early processors finish their work.) We are counting on statistics — it is even possible that with bad luck, a processor will suffer a conflict ten or twenty times before succeeding, but the probability of this is very small (10^{-10} or 10^{-20} respectively).

Fortunately, the problem of stragglers becomes less serious as the number of references before the barrier grows. Also, once the early processors reach the barrier, the switch load is significantly reduced through read combining, reducing the contention rate for the remaining processors. This suggests that the problem should be tuned to require many references between barriers if possible. At 200 references between barriers, this is a 15% effect on a 65,536-processor machine.

Size of the Problem

As we have examined dozens of programs for the Monarch, it has become clear that the Monarch is a powerful machine that must be applied to big problems. The summing example is almost too small to take advantage of the machine – it would have taken a single processor only one second to do the work alone. Imagine if we had examined how to sum, say, 1000 numbers on a 65,536-processor Monarch.

That example required each of the 65,536 processors to subtotal 15 numbers, requiring only 15 microseconds. In rough terms, the overhead was about 30 microseconds. This is consistent with results we have seen in other problems. Since program overhead runs in the tens of microseconds, one processor's piece of a computation should require 50 to 100 microseconds if the algorithm is to be efficient.

Fortunately, the user can configure the machine, via Mach, to have a more appropriate number of processors assigned to a task. The other processors are available to other users.

We have examined several dozen algorithms in detail for a 65,536-processor Monarch, Figure 21 plots the amount of parallelism we found versus the size of the problem.

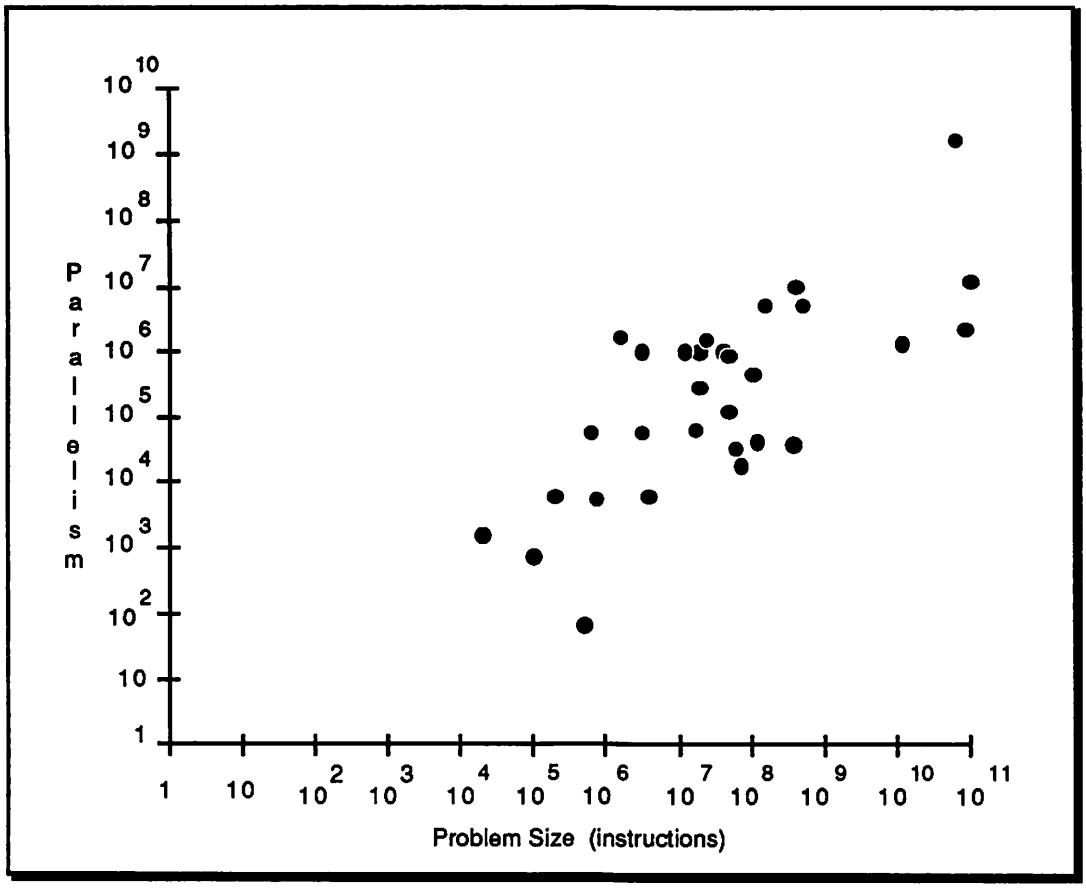

Figure 21 — Parallelism Found in Problems

Here *parallelism* is defined as the number of processors that resulted in 50% efficiency on the problem. More than 10,000 processors could be applied to all but six of the problems, and many could make use of a machine with more than 1 million processors. Generally, the larger the problem, the more parallelism was found. Most of the points are around a line where there are 100 instructions per processor, suggesting that the Monarch is appropriate for problems that can be divided into pieces with about 100 instructions each.

3.7 Software Summary

The goal of this section has been to illuminate the nature of programming the Monarch. In many ways, this parallel programming is familiar. The new knowledge and skills needed to program a parallel machine are within every good programmer's reach. Data and control structures, subroutines, and even program bugs are similar to programs for uniprocessors. The language extensions are minor and straightforward. Only two aspects of the programming are new. First, it is necessary to coordinate the processors from time to time

in order to ensure proper operation. Barriers, if used, are effective but crude. *Steal* offers a more difficult but also more elegant approach.

The second unique aspect is the performance impact of serial bottlenecks. In a massively parallel machine, these must be eliminated. This may require some effort.

Which problems are suitable for a massively parallel computer like the Monarch? The key requirement seems to be that they should require a large amount of computing. Massive machines need massive problems.

4. I/O System Design Issues

> While designing the IBM System/360, Gene Amdahl postulated two rules of thumb for a balanced system. The first rule related [processor] size with [main memory] size, stating that 1 byte of [memory] was required to support each instruction per second. The second rule related [processor] speed with I/O bandwidth, stating that one bit of I/O was required to support each instruction per second. [Siewiorek 82]

A 65,536-processor Monarch executes on average about 200 billion instructions per second. It has a trillion-byte virtual address space and a 137-billion-byte physical address space. Initially, the machine will have 16 gigabytes of main memory, but 4 and 16 megabit DRAMS will extend this to fill the physical address space. In memory size, the Monarch is well balanced.

On the I/O side, Amdahl suggests that we should have 25,000 megabytes per second of I/O capacity. It is hard to imagine what to connect to the Monarch at such a high rate. Very-high-performance video displays (1500 x 1000, 48 bit color, 75 Hz refresh) would refresh at 675 megabytes per second, and 37 of these would absorb this bandwidth. A disk system with 2000 very-high-speed disks (14 megabyte transfer rate) would also.

Just as the central processing portion of the Monarch achieves high performance through the use of many moderate-performance processors operating in parallel, the Monarch I/O system also achieves high performance by using a large number of moderate-speed, cost-effective I/O devices operating together. This matches the performance of an I/O device to our processor and provides an opportunity to achieve exceptional reliability in the I/O system.

A 65,536-processor Monarch has up to 2048 additional I/O processors; smaller machines have proportionally fewer. These I/O processors are identical to the other processors in the machine, and are connected to extra ports on the switch, but have a high-speed (4-wire) link connecting them to the I/O devices. Each I/O processor serves the I/O devices attached to it and communicates with other parts of the machine through the shared memory.

We estimate that each I/O processor can support a sustained bandwidth of about 2 megabytes per second. A machine with 2048 I/O processors would have a total I/O bandwidth of 4,000 megabytes per second, short of Amdahl's rule by less than an order of magnitude. We hope that in the compute-bound applications for which the Monarch is intended, this I/O bandwidth will be adequate.

The Monarch is being designed to support high-speed networks. Users access the machine through these networks. Although the network interface has not yet been designed, we envision our architecture connected directly to Hyperchannel and Hyperbus networks. In this way, any workstation on the network that has local graphics capabilities will be able to act as a graphics front-end between the Monarch and the user, using the X-windows standard.

Disk I/O Subsystem

The disk system of the Monarch will consist of about 200 small Winchester disks operating together. To maximize the peak transfer rate into and out of the machine, we dedicate one I/O processor to each drive. The system software of the machine performs reads and writes to the disk system in blocks. By spreading the data from a block over eight separate disk drives, we can increase the peak transfer rate to 16 megabytes per second for a single transfer and over 400 megabytes per second for the whole system.

Through parallelism we also have an opportunity to improve our tolerance of errors. Adding a ninth disk to each group of eight will let us ensure that the failure of any disk drive, any disk controller, any I/O processor, or any disk I/O system cabinet will not disable the machine. In fact, the machine will continue to run without interruption.

The data transferred to the ninth disk will be the exclusive-or of the data in all of the other sectors —a parity sector. That is, the first byte of the parity sector will be the bit-wise exclusive-or of the first byte in each of the eight data sectors, and so on. In addition to the parity sector, normal longitudinal error detection and correction will be provided for each sector.

If one of the disk drives crashes and must be replaced, no data is lost and the system keeps running. Since the drive is separately powered and cooled, the service person can slide it out of the disk cage and replace it with a new drive. The contents of this new drive can be replaced by scanning through the disk at some convenient rate, using the data on the other drives to generate the correct data for the new drive. Techniques like these are now becoming available in commercial disk systems.

5. Redundancy and Reliability

One of the primary advantages of a multiprocessor architecture is that with so many processors and memory modules, it would seem possible to survive the loss of some of the components without much effect on the overall machine. In case of a failure, it might be possible to provide uninterrupted service in some instances. In others, it will be necessary to stop the machine, reconfigure, and restart the operation. In any case, we should expect the machine to be highly available with little downtime.

In the design of previous machines, special attention has been paid to reliability and redundancy. In the Butterfly processor, for example, even the power supply is distributed throughout the machine so that power supply failures can affect only a small portion of the machine. In the Monarch, we have not reached that level of detail yet in the design, but we have addressed reliability in the architecture.

The most prominent failure modes are the ones that occur at the module level: processors, switch, concentrator chips, memory modules, and interconnect. In each case, the system design provides redundancy so that many single point failures can be tolerated. For example, interconnect failures can occur at several different points. At the boundary between each processor and memory board, 12 identical wires are available. If one or two of them are defective, it is still possible to access memory; the only effect is a slight increase in the conflict rate. It is easy to imagine that the machine would survive a failure of even hundreds of these paths. This applies to the connection between the concentrator chips and the memory modules as well. Similarly, it is possible that a connection between a switch chip and a concentrator chip will fail. Since every switch chip output port has two redundant paths to the same destination, a single point failure can be tolerated on any switch-chip-to-concentrator-chip path.

There are several options for dealing with the failure of a memory module. The one that we currently favor is to supply a spare module and put a small mapping table in each switch chip. The mapping hardware can be designed to permit rerouting of memory requests to spare modules in the course of the normal routing process.

Processor failure is perhaps the easiest to deal with in principle because it is simply a matter of excluding faulty processors from the pool of active processors.

6. Conclusions and Future Work

The Monarch is currently in the midst of implementation. At this moment the high-speed signalling regime is being tested with two micron switch chips. It has logged more than 30,000 device hours of operation at 125 megabits per second, passing over 10^{16} bits. The logic design of the processor is almost complete and simulated. The memory controller and concentrator remain to be designed. Although much engineering remains to be done, we feel the feasibility of implementation has been demonstrated.

We believe that the shared-memory computer architecture is the most flexible architecture currently under study. With it, many differing programming styles can be used efficiently. Previously, concern over the delay in accessing remote memory through a switch has discouraged consideration of shared memory for massively parallel architectures. We hope that in this paper we have shown a viable architecture that is both powerful and programmable.

7. Acknowledgments

The design and implementation of a computer system requires many good ideas, appropriate technology, and sufficient funding, but mostly lots of hard work by a talented team. The Monarch team has worked long and hard to reach the current stage, and we would like to thank them. The high-speed signalling technologies are the result of efforts by Paul Bassett and Lance Glasser (from MIT). Phil Carvey, Ray Tomlinson, Larry Dennison, Ken Sedgwick, Greg Fromen, Ward Harriman, Mike Fertsch, and Om Gupta have done much of the logic, VLSI, and packaging design and testing. The software analysis, simulator, and compilers were done by Charlie Selvidge, Allen King, Gregg Bromley, Bob Thomas, and John Robinson. The architectural design has been led by Will Crowther, Randy Rettberg, and Phil Carvey. Additional help in facilitating the project has been provided by Joe Wood, Nena Bauman, and Ed Walker. Randy Rettberg, John Goodhue, and Ben Barker have been the project leaders. Frank Heart and Dave Walden provided support and encouragement for the project. We also thank other members of the staff at BBN who have helped briefly, made suggestions, and provided encouragement.

The Monarch is not being designed in a vacuum, and we appreciate the ideas and comments of our computer system friends at other organizations.

Principal funding for the Monarch project is provided by the Defense Advanced Research Projects Agency and monitored by the Space and Navel Warfare office under contract N00039-86-C-0159. We thank the staff of this agency for their support.

8. References

[BBN 87] BBN Advanced Computers, *Butterfly Product Overview*, BBN Advanced Computers Inc., Cambridge, MA, 1987.

[Broomell 83] Broomell, G. and Heath, R.J., Classification Categories and Historical Development of Circuit Switching Topologies. In *ACM Computing Surveys* V. 15, No. 2, June 1983.

[Cohen 81] Cohen, D.and Lewicki, G., "MOSIS – The ARPA Silicon Broker," *Proceedings from the Second Caltech Conference on VLSI*, California Institute of Technology, Pasadena, CA, January 1981, pp. 29-44.

[Computer 87] Special Issue on Systolic Arrays, *Computer*, Vol. 20, No. 7 (July 1987), IEEE.

[Dubois 88] Dubois, M., Scheurich, C., and Briggs, F.A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, February 1988, pp. 9-21, IEEE.

[Fuller 78] Fuller, S.H. and Harbison, S.P., *The C.mmp Multiprocessor*, Report CMU-CS-78-146, Carnegie-Mellon University, October 1978.

[Gottleib 87] Gottleib, A., "An Overview of the NYU Ultracomputer Project," *Experimental Parallel Computing Architectures*, (Dongarra, J.J., ed.), Elsevier Science Publishers B.V., Amsterdam, 1987, pp. 25-95.

[Heart 73] Heart, F.E., Ornstein, S.M., Crowther, W.R., and Barker, W.B., "A New Minicomputer-Multiprocessor for the ARPA Network", *Proceedings AFIPS SJCC*, vol. 42, 1973, pp. 529-537.

[Hillis 85] Hillis, D. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.

[Jones 80] Jones, A.K. and Gehringer, E.F. ed., *The Cm* Multiprocessor Project: A Research Review*, Report CMU-CS-80-131, Carnegie-Mellon University, July 1980.

[Karp 87] Karp, A.H., Programming for Parallelism, *Computer*, Vol. 20, No. 5 (May 1987) pp. 43-57, IEEE.

[Knuth 73] Knuth, D.E., *The Art of Computer Programming, Volume 3/Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973.

[Kung 78] Kung, H.T. and Leiserson, C.E., Systolic Arrays (for VLSI). In the proceedings ofthe *Symposium on Sparse Matrix Computations and Their Applications*, (Knoxville, Tennessee) 1978 Academic Press, Orlando, Florida. Also in *Introduction to VLSI Systems*.

[Kung 82] Kung, H.T., "Why Systolic Architectures?" *Computer*, Vol. 15, No. 1 (January 1982) pp. 37-46, IEEE.

[Pfister 85] Pfister, G.F., Norton, A., "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Computer Society Press, August, 1985, pp. 790-797.

[Pfister 87] Pfister, G.F., Brantley, W.C., et al., "An Introduction to the IBM Research Parallel Processor Prototype (RP3)," *Experimental Parallel Computing Architectures,* (Dongarra, J.J., ed.), Elsevier Science Publishers B.V., Amsterdam, 1987, pp. 123-140.

[Rashid 86] Rashid, R.F., "Threads of a New System," *UNIX Review*, August, 1986.

[Rettberg 86] Rettberg, R.D., Thomas, R., "Contention Is No Obstacle to Shared-Memory Multiprocessing," *CACM,* ACM, December 1986, pp. 1202-1212.

[Seitz 85] Seitz, C. The cosmic cube. *Communications of the ACM* 28, 1 (Jan. 1985), 22-33.

[Siewiorek 82] Siewiorek, D.P., Bell, C.G., and Newell, A., *Computer Structures: Principles and Examples,* McGraw-Hill, New York, 1982, pp. 46.

[Thomas 86] Thomas, R.H., "Behavior of the Butterfly™ Parallel Processor In The Presence Of Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing*, IEEE Computer Society Press, 1986, pp. 46-50.

[Wulf 78] Wulf W.A. and Harbison, S.P., *Reflections in a Pool of Processors*, Report CMU-CS-78-103, Carnegie-Mellon University, February 1978.