7-1-1984

Simulated Performance of a Reduction-Based Multiprocessing System

Robert M. Keller
Harvey Mudd College

Frank C. H. Lin

Recommended Citation

Keller, Robert M., and Frank C.H. Lin. "Simulated Performance of a Reduction-Based Multiprocessing System." Computer 17.7 (July 1984): 70-82. DOI: 10.1109/MC.1984.1659188

Multiprocessor systems present unique concurrency problems. Rediflow combines disciplined von Neumann processes with a hybrid reduction and dataflow model in an effective packet-switching network.

Simulated Performance of a Reduction-Based Multiprocessor

Robert M. Keller and Frank C. H. Lin, University of Utah

Multiprocessing systems have the potential for increasing system speed over what is now offered by device technology. They must provide the means of generating work for the processors, getting the work to processors, and coherently collecting the results from the processors. For most applications, they should also ensure the repeatability of behavior, i.e., determinacy, speed-independence, or elimination of "critical races."[1-6] Determinacy can be destroyed, for example, by permitting—in separate, concurrent processes—statements such as "$x := x + 1$" and "if $x = 0$ then . . . else. . .", which share a common variable. Here, there may be a critical race, in that more than one global outcome is possible, depending on execution order. But by basing a multiprocessing system on functional languages, we can avoid such dangers.

Our concern is the construction of multiprocessors that can be programmed in a logically transparent fashion. In other words, the programmer should not be aware of programming a multiprocessor versus a uniprocessor, except for optimizing performance for a specific configuration. This means that the programmer should not have to set up processes explicitly to achieve concurrent processing, nor be concerned with synchronizing such processes.

Language and concurrency

Programs expressed in functional languages possess a fair amount of implicit concurrency. The conceptual execution of a functional program is based purely on the evaluation of expressions, not on the assignment of values to memory cells. Accordingly, there can be no "side effects" of one function on another, which ensures determinacy; a program gives the same results regardless of the physical aspects of communication between processors or the number of processors involved in its execution. These languages seem to be ideal for the programming of multiprocessors when distinction between them and uniprocessors is undesirable. Functional languages also have other conceptual advantages that have been discussed elsewhere.[7-9]

To demonstrate how a functional language provides for concurrent execution, consider an expression such as

max[subexpression-1, subexpression-2]

where *max* is the usual numeric maximum function (or any other function which requires both of its arguments). A concurrent execution model carries out three important aspects:

(1) Spawning of tasks to evaluate the two subexpressions concurrently;
(2) Synchronization to determine that both subevaluations are complete; and
(3) Evaluation of the maximum, once completion is established.

Obviously, only the third of these aspects would be found in a sequential implementation; the first two are implicit in a concurrent functional implementation. In contrast, the specification of these mechanical aspects is often explicitly required in process-oriented languages.

Generating concurrently processable work can be amplified through appropriate data structuring. For example, in many functional languages, an expression can be sequence-valued, where a sequence is represented as a list, array, or tree. Through the use of operators such as *apply-to-all*, here designated as \\, a similar expression can be used to apply a function, such as *max*, aligned pairwise to the components in two sequences:

max\\([1, 3, 5], [6, 4, 2]) = [6, 4, 5].

Applying *max*\\ to a pair of sequences of length *n* could thus generate *n* independent tasks for concurrent execution. Further, if there are unevaluated subexpressions in those sequences, additional tasks could be generated to evaluate the subexpressions themselves. Function *max* could be replaced with much more complex functions.

Such implicit concurrency is exploitable through a functional language but not in languages such as Pascal, since evaluating the arguments to a function in them can have the side effect of modifying parameters or global data. Then, because of the order in which such side effects might occur, the behavior is not generally repeatable.

When encapsulated, local side effects provide one way of dealing with many distributed local states. In fact, totally encapsulated, sequential programs with only local side effects may be considered semantic abbreviations for a restricted form of a functional program.[10]

Once an appropriate functional framework is built, indeterminate constructs can be accommodated. For example, we have shown how the simple extension that allows the indeterminate "merge"[11] can be used to augment a functional language and ensure serializability in distributed database applications, including concurrent updating.[12]

It is also possible within a functional framework to assign subprograms to processor and memory resources. For example, a "site pragma" can force the execution of particular subtransactions of a database system on particular sites.

Concurrent evaluation models

Four categories of evaluation model are available, with varying degrees of facility, for getting work to processors and collecting results coherently.

Multiple processes with shared memory. The notion of a "process" is an abstraction of the execution of a program for a von Neumann computer. A process obeys a sequence of commands, each specifying an assignment to a register, a test, etc. The earliest concurrent computation models were based on spawning several such processes within a common memory space.[13] Communication between processes involved inspecting a register to which another process had assigned a value. To make such communication somewhat coherent, a variety of synchronizing constructs were invented.[14]

Multiple processes with message-passing. To eliminate sources of indeterminacy or isolate their effects, some systems forbid general sharing of memory locations. They employ message-passing as the fundamental means of communication. In such schemes, one process specifies a message to be sent to another, either by naming the other or by naming a common linking channel.

Certain disciplines can be imposed on a message-passing system to guarantee determinacy. For example, determinate behavior is guaranteed[6,10] if

(1) a process, once it decides to examine an input message buffer, is committed to wait for a message to be there;
(2) no two processes can share a common input buffer; and
(3) no two processes can share a common output buffer.

Put another way, a collection of conventional processes has an overall functional behavior, provided the above criteria are met. This functional behavior is used in a limited form to connect Unix processes via pipes,[15] which can be viewed as a special case of functional composition. Other functional programming systems attempt to exploit this phenomenon in a more general form.

Dataflow. Dataflow computers[16,17] use message-passing in small decomposable units of work. Typically, each primitive operator is really a "process" performing the same operation time after time on streams of values. Dataflow machines attempt to eliminate the overhead that would accompany explicit sequential processes.

A simple extension can augment a functional language and ensure serial database distribution and concurrent updates.

Programs in dataflow machines are often represented as directed graphs, with the nodes representing operators and arcs representing message queues. Message queues are assumed to have a one-message capacity. If one wishes greater asynchrony, which would be necessary for maximal concurrency when processes are generating messages at widely varying rates, additional identity operators can be introduced to balance the processing rates. These identity operators provide more buffer stations, through which messages must pass to get from one node of the original graph to another. Unfortunately, there is no algorithm for balancing a cyclic program in which the number of iterations is data dependent.

Evaluation by graph reduction. In order to achieve maximum asynchrony, it is helpful to decouple the production of values from their consumption as much as possible.

A graph reduction model provides one means of decoupling. In particular, it provides an alternative to bounded buffering by using a linked list with cells drawn from a global storage pool. The reduction model of computation provides an elegant way of achieving such an effect.

In the reduction model, work is spawned at a finer granularity than processes. Specifically, task granularity corresponds to that of function calls in conventional languages. Tasks might typically perform a few arithmetic, logical, or structuring operations, including storage allocation for portions of data structures. However, rather than having a long-term sequential behavior, they would continue by generating other such tasks. Thus, a rapidly dividing computation can be represented by a task which generates two other tasks, while the equivalent of a sequential computation can be represented by a task that does some work, spawns another task, then dies.

Task spawning can be illustrated in programs with a functional syntax similar to one suggested by Burge.[7] One can compute the factorial function by the "divide-and-conquer" strategy as follows:

```
Factorial(x) = DAC(1,x)
   where DAC (m,n) =
      if m = n
         then m
         else DAC(m,med) * DAC(med + 1,n)
            where med = (m + n) /2
```

In the reduction model, a task and its supporting storage are allocated each time an instance of DAC is demanded. For example, factorial 100 demands an instance DAC(1, 100). A given instance, DAC(m, n) would either terminate more or less immediately if $m = n$ or demand two more instances, as indicated in the definition. When an instance of a function is computed, the value replaces the instance itself. This is proper, since in functional languages, a function with particular arguments can always be replaced with the corresponding value without loss of generality. Thus, if the instance is shared, all sharers will benefit from one computation of the value.

The second example shows process-like behavior in the reduction model. Suppose we want a process to compute the sequence

$$1 \,\hat{}\, 2 \,\hat{}\, 3 \,\hat{}\, \ldots$$

where ˆ denotes succession, read "followed by." The following function, NUMS_FROM describes a process that computes the numbers from its argument n on

$$\text{NUMS\_FROM}(n) = n \,\hat{}\, \text{NUMS\_FROM}(n+1)$$

In the reduction model, a demand for NUMS_FROM(n) would generate a data structure containing n followed by an instance of NUMS_FROM($n+1$), as shown in Figure 1. However, that instance would lie suspended[18] until it is demanded. This is the analog of a *producer* process, which blocks until more data is requested. As more of a sequence is demanded, more of the structure is generated. If many such demands are generated rapidly, the evaluator simply lays out a structure for receiving the result values once they are computed.

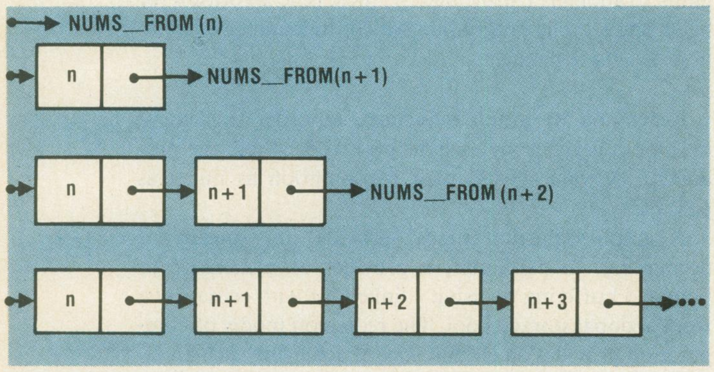


Figure 1. Computing a sequence by reduction. Arrows are conventional pointers in a von Neumann memory.

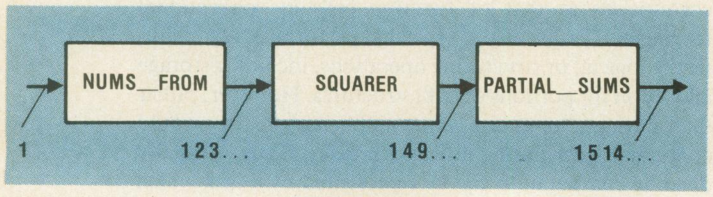


Figure 2. Pipeline interconenction of processes.

Similarly, we could define a *transducer* process, for example, a SQUARER that squares each element in a sequence of numbers:

$$\text{SQUARER}(x) = \text{HEAD}(x)**2 \,\hat{}\, \text{SQUARER}(\text{TAIL}(x))$$

where HEAD and TAIL are defined by HEAD($a \,\hat{}\, y$) = a and TAIL($a \,\hat{}\, y$) = y.[8,18-20]

The function PARTIAL_SUMS below is a function of its input sequence, which eventually depends on every element of that sequence for its output.

$$\begin{aligned} \text{PARTIAL\_SUMS}(x) &= \text{AUX}(x, 0) \\ \text{where AUX}(x, ac) &= b \,\hat{}\, \text{AUX}(\text{TAIL}(x), b) \\ \text{and } b &= ac + \text{HEAD}(x) \end{aligned}$$

The "state" in this case is represented as the second argument (which serves as an "accumulator") to the auxiliary function AUX. Incidentally, this example counters the myth that functional programs are incapable of modeling "state" or "history-sensitive" operations.

Recalling that pipe connections of processes are functional compositions, we observe that, if we connect our three functions NUMS_FROM, SQUARER, and PARTIAL_SUMS together in a "pipeline," as shown in Figure 2, the meaning of functional composition gives us exactly the intuitive behavior: The output of the pipeline is the sequence of partial sums of the sequence of squares, beginning with $n**2$.

To compare the reduction and dataflow models, we also feed the output of function SQUARER into a second function, POLY, which, let us say, computes some complicated function of each input, such as a large-degree polynomial. Here PARTIAL_SUMS and POLY would not be expected to consume the stream at the same rates. Figure 3 illustrates storage cells that might be allocated in a particular reduction computation of the functions above. Obviously, it is hard to predict in advance how much buffer space should be allocated for the output of SQUARER, so the dynamic allocation scheme provided by the reduction model is, therefore, useful in relaxing the constraints imposed by a bounded-buffer dataflow implementation as illustrated in Figure 4.

Combining reduction and dataflow. Each of the reduction and dataflow models (also called "structure" and "token" models, respectively[21]) has certain disadvantages. As we have seen, the reduction model is useful where asynchrony requires unpredictable buffering or for rapid task division with recursion. On the other hand, when stream-based communication does not require great asynchrony, the dataflow scheme has the advantage that no storage allocation is required. Dataflow models that "unfold," such as the "U-interpreter,"[22] approach the unwinding feature of the reduction model within a dataflow model.

One shortcoming of the reduction model is inefficiency in the implementation of essentially sequential computations. Here the reduction model uses "tail recursion," as in the function SQUARER above, and allocates a new task for each iteration of a loop. If the consumption of SQUARER's output is sufficiently slow, there is no need for the unwinding effect provided by reduction. It is more efficient to compute SQUARER sequentially. However, with disciplined message passing, it is possible to integrate such sequential computations in the context of the reduction model. The technique permits von Neumann code to be encapsulated into the node of stream processing functions.[10]

This is the approach taken to introduce dataflow behavior into the Rediflow system, as we will describe in the next section. In Rediflow, we can use the pointers present in the structures of the reduction model to provide logical channels on which tokens flow. Use of the pointers provides a convenient way of setting up dynamically generated dataflow graphs. Such functions can be combined arbitrarily to build more complex systems and can be interfaced with corresponding systems implemented by pure reduction. Two simple functions can interface a reduction-implemented function with a dataflow-implemented function—one produces a stream of tokens from a structure and the other builds a structure from a stream of tokens.* The following simulation comes from a pure reduction subset of the evaluator. Efficiency of examples containing large, essentially sequential components will be improved by the proposed Rediflow integration.

*Further details and uses of this construct are treated by Tanaka.[23] We also describe an overall approach to the corresponding language constructs.[24]

Rediflow system organization

"Rediflow" is the name we give to our function-based concept for multiprocessor system design and attendant software capabilities. The name is a combination of the words "reduction" and "dataflow," two models for concurrent evaluation described above. Our model also includes disciplined aspects of the von Neumann evaluation model. Having justified the functional approach, we now turn to issues of physical organization.

Hardware issues. The main problem in assembling processors for multiprocessor execution is to distribute work effectively while avoiding extensive communication overhead. In general, links between processors and memories must be provided, and programs expressed in appropriate computer languages must be mapped onto the resulting system. Device technology remaining invariant, it is the ease in mapping that determines the success of a multiprocessing system.

The ease of mapping depends on the class of applications, the languages used, compilers, and the underlying hardware configuration. Clearly, for a fixed application, a

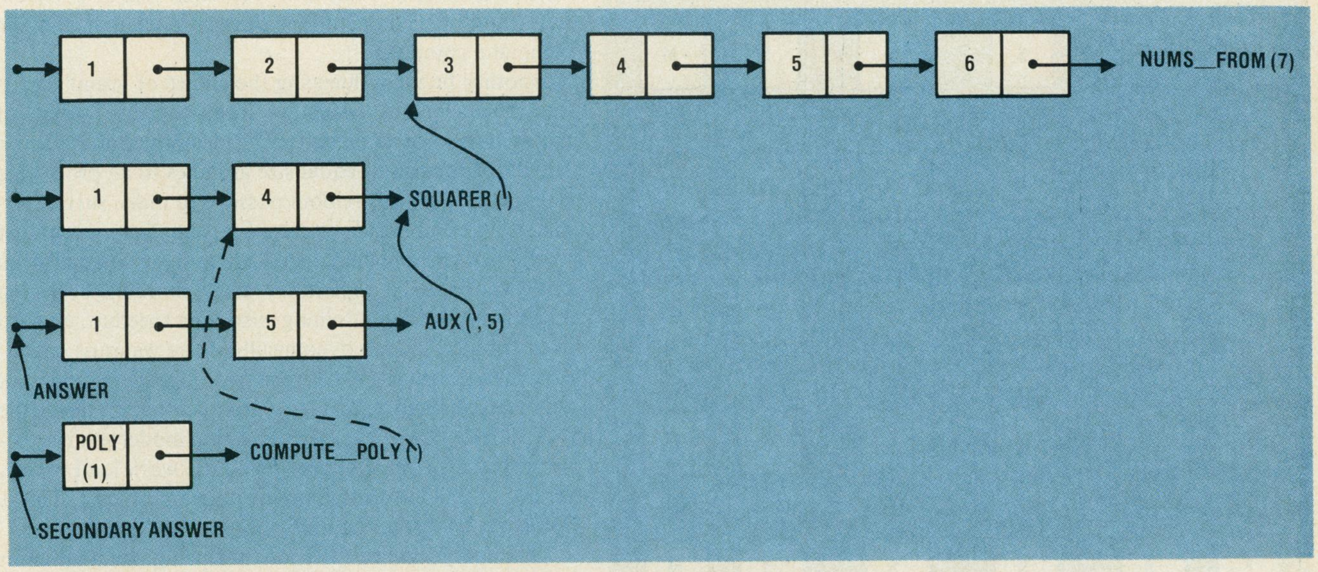


Figure 3. Reduction implementation of pipelining.

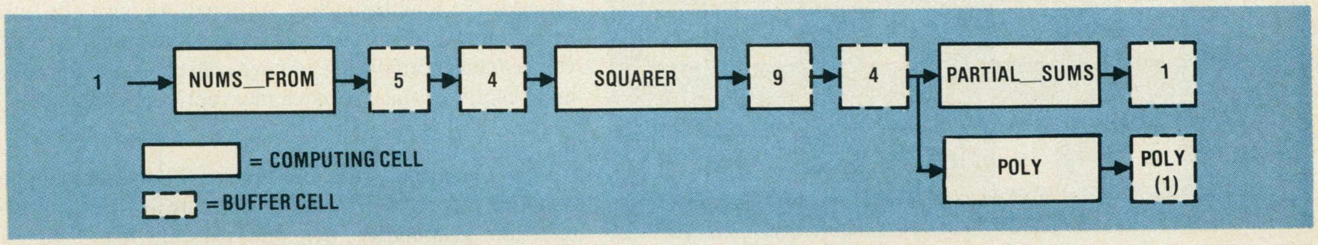


Figure 4. Dataflow implementation of pipelining.

special-purpose machine can be designed to out-perform all others on that application. Our interest is not in such machines, but in the techniques that exploit multiprocessing power for a wide range of applications. To delimit this range, it is useful to group applications, according to regularity, size span, and granularity.

Applications of high regularity contain many very similar operations with similar computational demands. Here, approaches such as vector processors, cellular arrays, or static dataflow may be appropriate.[16] The size span characteristic of a set of applications relates to the extent the problem size is apt to vary over the lifetime of the system. While a particular array processor may be ideal for problems that can be contained in one array load, there may be difficulties in folding or decomposing larger problems to match the processor configuration and maintain acceptable performance. We have designed Rediflow to accommodate problems that lack such regularity, but that may have very large size spans.

Fine-grain operations would be those at the level of bit or arithmetic operations. Large grains would be processes or entire jobs. Rediflow is aimed at applications appropriate for medium or function-level granularity and larger. Irregularly structured problems, such as knowledge-base systems, should be appropriate for exploiting this level of granularity: These problems contain strands of operations that must be done sequentially and are therefore of rather coarser grain than simple arithmetic operations, but of finer grain than many typical processes. Other applications of medium granularity are certain adaptive numerical calculations and certain types of signal processing. Rediflow is also aimed at combinations of several application areas that interact in unpredictable ways.

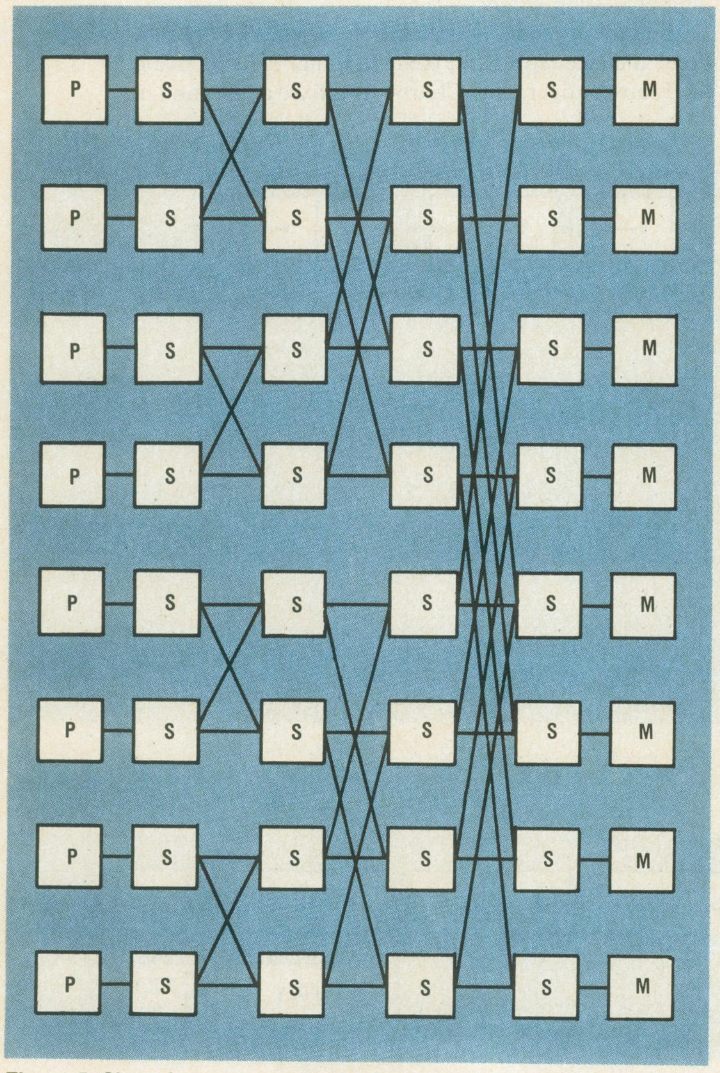


Figure 5. Shared-memory multiprocessor configuration.

Implications of granularity. Two areas of trade-off in granularity are communication overhead and flexibility in load balancing. Systems may fail to exploit their peak capacity because of excessive data communication between granules. This form of communication does not exist in the equivalent purely sequential computation but may become significant if there are several processors. For very small grains, the delay due to communication may exceed the delay of the operations themselves. For this reason, small granularity is exploitable only if the regularity is high enough that necessary communication paths are relatively short and static or if there is little communication between grains. Wide, indiscriminate distribution of many small grains increases the likelihood that communication overhead will be large. Rediflow clusters several small operations together inside one function body, obviating their distribution.[20] Thus, the execution rate of a sequential strand of such operations can approach that of a von Neumann computer.

Another factor influencing the choice of granularity is load balancing, by which we mean the distribution of grains to the processing units. The ideal situation is a single initial distribution of equal-size granules to all processing units. However, it is seldom possible to make such determinations beforehand, because many applications present work loads that are data dependent and not susceptible to static analysis. To exploit the available multiprocessing resources fully, thus attaining maximum speedup, we need to distribute the load dynamically. Here we must pay attention to the trade-off between small granules that permit a more even balance, and large granules that minimize the total distribution effort. The reduction model seems to offer sufficiently fine grain to spread the work load widely, yet not so fine as to entail undue distribution cost. This is borne out by our initial simulations.

This study concentrates on balancing medium-grain tasks that occur in reduction evaluation. The embedded dataflow processes described on p. 73 are large grained. Their advantages must be weighed against the extra complications in load balancing. Although we have not fully investigated this issue, it appears that large-grain processes can be rebalanced during garbage collection, but the costs and benefits of this approach have not been analyzed.

Interconnection issues. Multiprocessing systems can be classified by processor-memory structure as well as by application granularity. At one extreme are "shared memory" configuration (see Figure 5, which shows processors and memories separated by a switching network). They appear to provide any processor a uniform access time to any memory. However, this uniformity may be compromised if there is significant contention at individual switches. Unfortunately, the uniform delay also becomes uniformly longer with increasing numbers of processors and memories. It is possible to introduce caches that are coupled closely with processors and retain local information for faster access. However, caches also introduce the difficult problem of "coherence"[25,26]; when one processor updates information cached by another, the latter must be invalidated—at a cost of additional communication overhead. Any machinery introduced to overcome this problem further dilutes the useful capacity of the system.

At the other extreme, in the Rediflow configuration and in others,[27] each processor is closely paired with a memory, and a network of packet switches is used to communicate between these pairs. Although sometimes called a "loosely coupled" system, the coupling is actually very tight as far as a single processor-memory pair is concerned. Certainly, the peak bandwidth in such a system consisting of n pairs is much higher than one with n processors and n memories separated by a large switch. The following points may be noted about the paired configuration:

- The worst-case delay is not attained for every processor-memory reference; many delays will take at most the time of one local access, and on the average, the delay will be less than worst-case.
- The configuration permits the exploitation of "locality,"[28] which means that logically related operations can cluster their references to a subset of data. In Rediflow, function-level granularity permits a certain degree of such clustering.
- The coherence problem is avoided, since each processor has exclusive control over its own memory. This exclusiveness also obviates introduction of special instructions for multiprocessor memory access, such as test-and-set and its derivatives.[29]
- The overall cost for switch hardware grows linearly with the number of processors (rather than as $O(n \log n)$ or worse) because every hardware component used for switching has a processor associated with it.

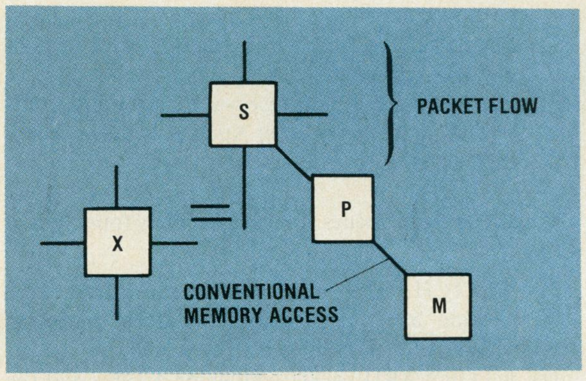


Figure 6. Sketch of an Xputer.

In Rediflow, we call the combination of a processor-memory pair and a packet switch for information transfer an *Xputer,* a term suggested by its similarity to the "transputer" chip announced by INMOS.[30,31] A conceptual sketch of the Rediflow Xputer information flow is shown in Figure 6.

The system-level aspects of Rediflow permit a wide variety of interconnection networks, including the shuffle-exchange shown in Figure 7, the grid in Figure 8, or various cube configurations.[9] The following minimal assumptions are all that are necessary for effective operation:

- Addressability: There must be a means for uniquely addressing any memory location in the entire system so links between functions concurrently executing in different Xputers can be dynamically established.
- Routability: Given a request to fetch from or store in a specified location, the switch can determine where to route the request once the links have been established.

Links are purely virtual, as implied by pointers; there are no dedicated logical paths.

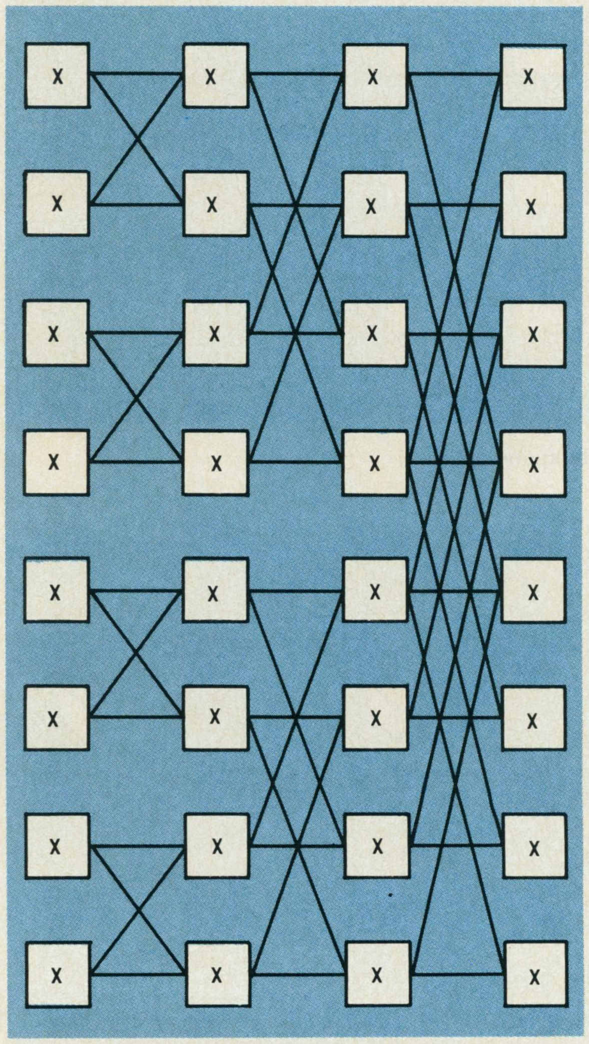


Figure 7. Shuffle-exchange Xputer network.

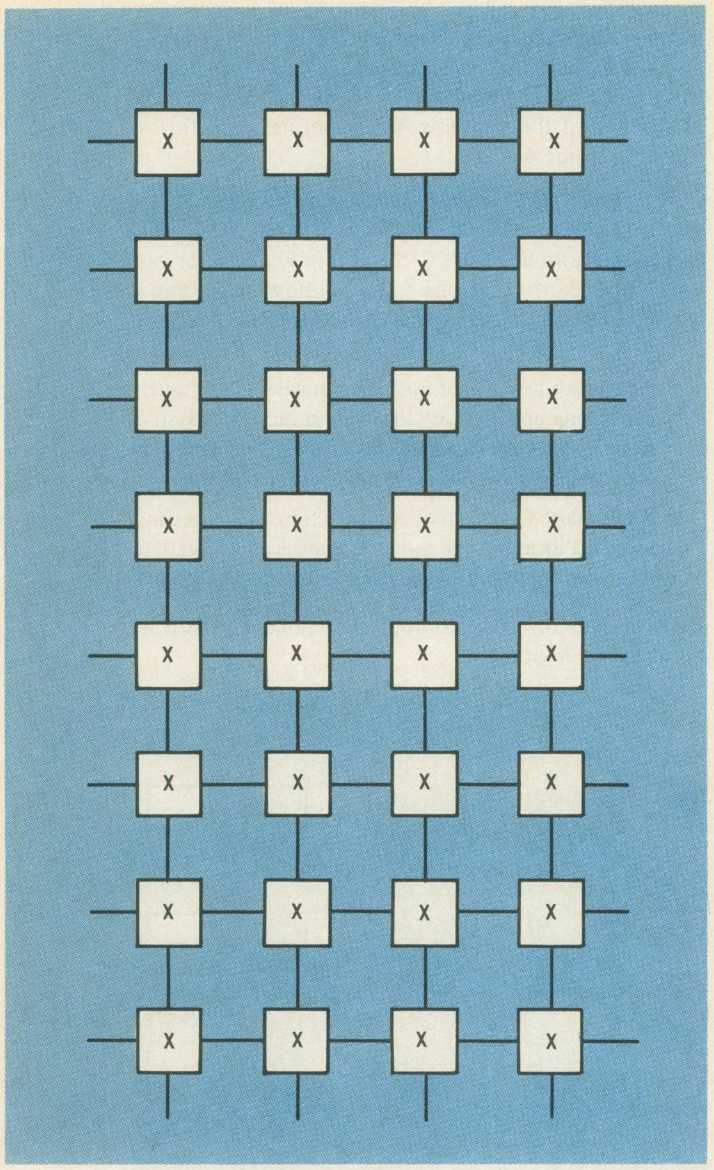


Figure 8. Grid Xputer network.

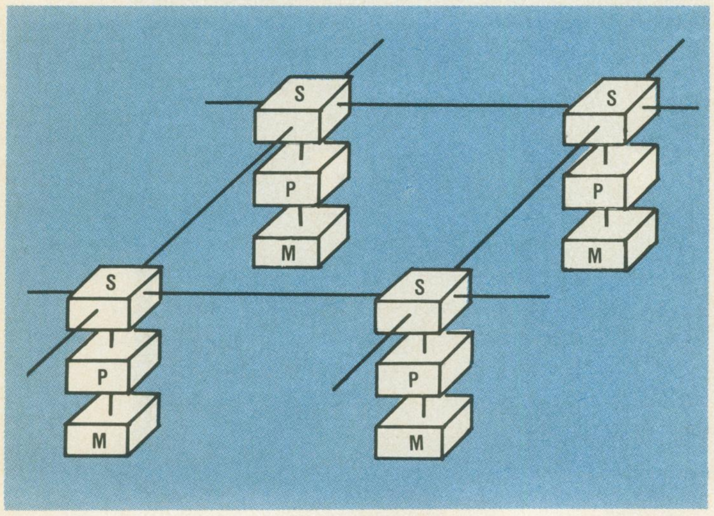


Figure 9. Layered view of grid network.

Switches, processors, and memories, however, can be taken to form logically parallel layers, as indicated by Figure 9. Interconnection between Xputers exists only at the switch layer, while the memories in the memory layer have a combined global address space. If one Xputer needs to access the memory of another, it forms a request packet containing the address to be accessed. That packet is then routed within the switch layer to the Xputer containing the addressed location. A result packet is eventually formed; it is then routed to the requesting Xputer. This request/return mechanism is integrated with the demand/drive mechanism of reduction evaluation, so that remote invocation of functions can take place.[20]

Input/output devices, which are not shown, may be attached at any nodes. Sequential I/O devices are interfaced through von Neumann processes, as mentioned earlier. There is also no conceptual difficulty in including multiple, secondary storage devices. Addressable devices can be used to implement a virtual memory mechanism as an extension of the system-wide address space.

Figure 10 indicates how a data structure appears when spread over several Xputers. The pointers can serve conventionally as references, but they also define logical channels to implement the dataflow aspect of Rediflow.

Load distribution and balancing. To avoid bottlenecks when the system is scaled, we eschew a centralized queue from which idle processors get their work. Instead, the method for migrating work is itself distributed. To avoid granularity so small that communication delays become significant, we use medium-grain function invocation tasks as units of migrable work. As shown, for example, in the definition of SQUARER, process-like behavior can be implemented by such invocations; thus, the resulting process becomes *mobile.* To describe further load distribution, we must show the mechanism that permits task migration and the mechanism that causes it.

Linkage mechanics. As with most multiprocessor organizations, the backlog of work is held in one or more queues. In Rediflow, there are four queues per Xputer:

(1) The IN queue of incoming FETCH requests for location contents and acknowledgments of such requests (called FORWARDs).
(2) The APPLY queue of migrable function application tasks. The mapping of interfunction into the global address space permits items on this queue to be moved at will to other Xputers.
(3) The LOCAL queue, which contains units of nonmigrable fine-grain work.
(4) The OUT queue that receives FETCH and FORWARD requests for other Xputers or the acknowledgment of such requests.

The items on these queues are represented by four types of packets:

(1) A FETCH packet, which contains the address of the location being fetched and the storage location where the fetched value is to be stored.
(2) A FORWARD packet containing a value and the address into which the value is to be stored. This

location contains appropriate information to notify a suspended task waiting for the stored value. Forward packets are generated in response to earlier fetch packets.

(3) An APPLY packet containing
  (a) a closure—a record with a pointer to code for evaluating the function to be applied and a pointer to a tuple of import values for that function (the latter correspond to the values of free variables in the lambda-calculus sense); and
  (b) an argument for the function being applied, which might be a pointer to a tuple, for example, if the function is viewed as multiargument.

(4) Packets on the LOCAL queue are simply addresses in the local memory of operation nodes that need to be evaluated; they serve as the analog of "instruction pointers" in a conventional computer.

Because all result data has preallocated globally-addressable locations, it is not necessary to use any form of token matching to get the data to its destination.[17] Fast von Neumann-style addressing is exploited in each Xputer, as well as among all Xputers. Addressing permits the switch to provide the shortest possible route. This style of routing is taken directly from the earlier conception of AMPS.[21] It is inaccurate to suggest that the routing technique is a form of "token matching."[32]

A limited form of content addressing may be ultimately required to cache pure copies of function code in an Xputer, following an initial fetch from secondary or resident storage. This addressing is necessary only if the number of different function codes in a run is too large for a directly-indexed cache.

There is no correspondence between logical code and specific Xputers. The same function may be executed simultaneously in many different Xputers, and one Xputer may be multiplexing the execution of many functions. For example, one function may be suspended while it waits for data fetched through the network.

Loading mechanics. We view the migrable tasks as molecules of fluid poured over the switch layer. The mechanics of migration are easily described by the notion of *pressure,* which forces the fluid to move among Xputers. The internal pressure of an Xputer indicates how busy the Xputer is—in other words, its availability for additional work. In the present model, the only contributions to internal pressure are the number of packets on an Xputer's LOCAL and APPLY queues and the fraction of memory occupied. The latter is important because the reduction model basically relies on a dynamic memory allocation system. The function currently used is

$$\text{internal pressure} = \text{length of queue} + c\left(\frac{1}{1 - \text{fraction of memory occupied}}\right)$$

where c is a constant. This function minimizes the contribution of memory occupancy until it is nearly full.

Our distributed load-balancing technique involves Xputers furnishing pressure information to one another in an effort to determine where to route excess backlog. However, it is not enough for an Xputer to furnish only its internal pressure to others. If this data were sufficient, a heavily loaded Xputer could be surrounded by a wall of nominally loaded ones and not be aware that, outside the wall, there were Xputers that could accept some of the extra load. Therefore, we introduce the notion of *propagated* pressure, which is what an Xputer indicates to its immediate neighbors. The propagated pressure of an Xputer is a function of both its own internal pressure and its external pressure, which is in turn a function of the propagated pressures of its neighbors.

When an Xputer's internal pressure exceeds the external, some packets from its APPLY queue may pass into the interconnection network, where they are distributed to Xputers with lower pressures. Rediflow employs a switch capable of directing packets along pressure gradients to find such low points. When a packet reaches an Xputer with a local pressure minimum, it is absorbed into its APPLY queue. This absorption tends to raise the pressure of that Xputer and lessen the likelihood that it will receive more packets until completion of work reduces its internal pressure.

Obviously, pressure of Xputers is continually changing. Accordingly, it is necessary to update each Xputer's sense of its external pressure frequently. Updating requires that an Xputer send out a packet containing data on its propagated pressure when a sufficient pressure change has occurred.

One heuristic function that works moderately well is to define the propagated pressure in terms of the equations

$$\begin{aligned} PP(X) = \text{if } & PI(X) < \text{threshold} \\ & \text{then } 0 \\ & \text{else } \min[1 + PE(X),\ \text{ceiling}] \end{aligned}$$

where PP, PI, and PE are, respectively, the propagated, internal, and external pressures, and threshold is a settable parameter. For ceiling, we use 1 + the diameter of the network (the length of its longest path which does not include any node twice), and for PE we use

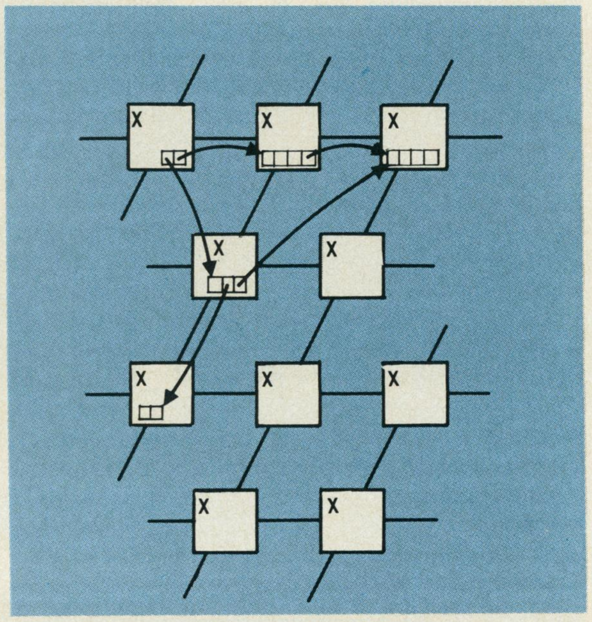


Figure 10. Spreading of a data structure over an Xputer network.

$$PE(X) = \min\{PP(Y) \mid Y \text{ is a neighbor of } x\}$$

The above function permits packets to flow toward a minimally loaded node. In fact, $PE(X)$ can be shown to give the number of links to be traversed to reach such a node. The "continuous" computation of $PP(X)$ for each X is a form of "relaxation" and, of course, need not be precise, since the load is constantly changing. The effort involved is small enough to be integrated into an intelligent packet switch.

Saturation effects. The phenomenon of saturation occurs when all Xputers are so busy that any attempt to migrate apply packets would be futile, despite an extreme internal/external differential. The Rediflow load-balancing mechanism uses a ceiling on the value of propagated pressure to detect saturation. When the external pressure of an Xputer reaches the ceiling, migration ceases. This mechanism enhances locality, since a greater proportion of local allocation implies a greater proportion of local memory references.

As mentioned earlier, the reduction model of computation offers the advantage of easily spawning concurrently executable work for migration to other processors. In effect, a "spanning tree" is grown. The tree corresponds to a single growing and shrinking expression; the "output" of the running program may be extracted on a continuing basis. The default mode of servicing each Xputer's APPLY queue is FIFO, which generates the tree breadth-first and thus reaches concurrently executable nodes earlier. To prevent the generation of additional work during saturation, an Xputer switches to LIFO for depth-first generation in order to throttle its rate of packet production. This constraint reduces queue overflows and overcommitment of memory space, which could result in a kind of deadlock (this technique was also observed by Burton and Sleep.[31]) In saturated mode, operators that normally demand arguments concurrently must demand them sequentially. This change turns out to be easy in our reduction implementation, which also allows introduction of program-control mechanisms for reducing the possibility of overcommitment of resources.[33]

General packet flow. A schematic overview of the organization of an Xputer appears in Figure 11. This diagram assumes that pressure-sampling information is sent through the switching layer in the form of pressure packets intermingled with other types of packets (APPLY, FETCH, and FORWARD). This assumption is used in our simulation results, except for garbage collection, which is not yet simulated on a packet basis.

Performance evaluation

The performance of the Rediflow architecture is now being evaluated in simulations. We currently use an introspective model, that is, one in which speedups are measured against a single processor with the same technological elements, architecture, and evaluation model. After certain improvements, we hope to run simulations to calibrate performance against existing machines.

General simulation technique. The simulator for Rediflow implements a distributed interpreter for the graph-reduction model, including embedded von Neumann processes. Delays inherent in the switch layer are parameterized

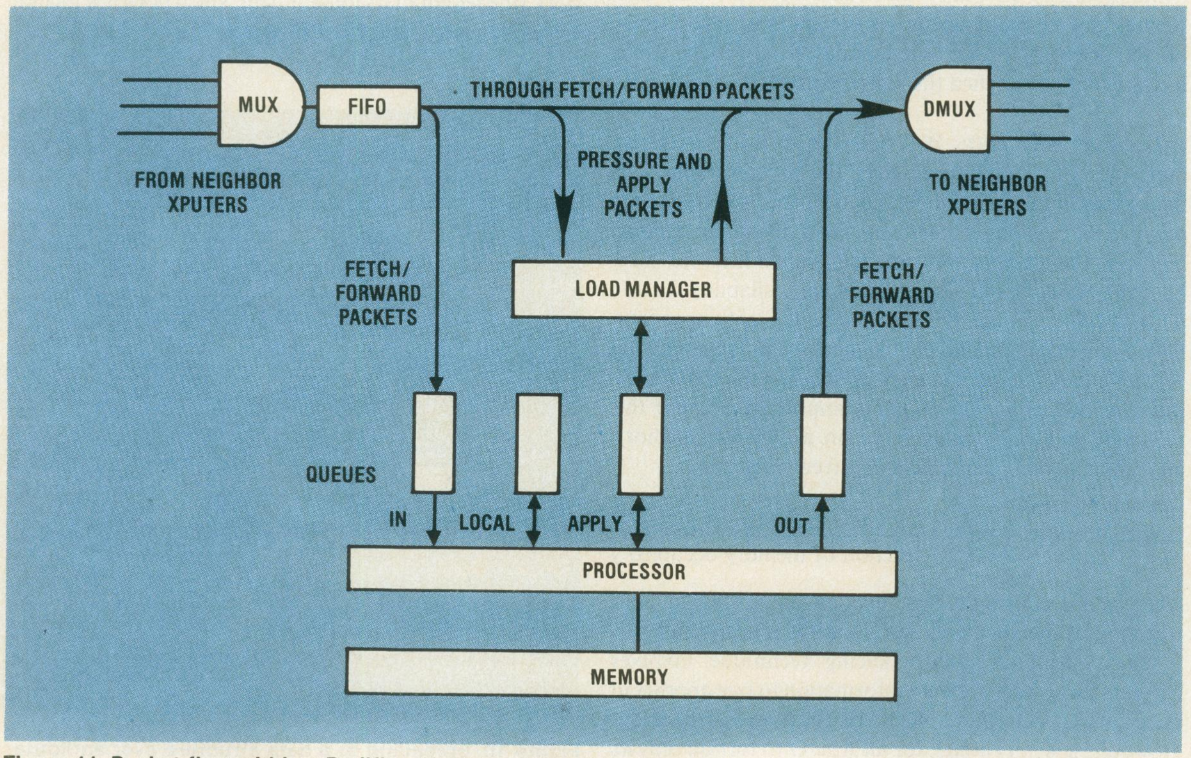


Figure 11. Packet flow within a Rediflow Xputer.

relative to processor delays. All buffers described for the Xputer are simulated. Also, a FIFO input buffer for each switch is included for resolving contention.

We assume that each graph-reduction step and primitive operator computation takes a single time unit, which is set to an estimated average value. We do not yet simulate the distribution of code, in effect assuming that a copy of the pure code of every function is present in each Xputer memory. Since caching of code is intended and the number of distinct code blocks is usually small in comparison to the number of block instantiations, simulation of code distribution would change performance by at most a small additive constant.

Simulations based on the "ply" notion provide a basis for evaluating Rediflow concurrency in combinations of synchronous and event-driven processes.

Garbage collection is accomplished by concurrent compactions[34] in each Xputer. We intend eventually to have a message-based,[35] global compaction system for coordinating compaction. We do not yet simulate distributed, concurrent garbage collection because we have so little memory within which to simulate and collection would consume an unrealistic proportion of time. We copy without shifting data from one Xputer to another to avoid spoiling the spread of data needed for concurrent execution.

The simulation is a combination of synchronous and event-driven methods. Processor and switch cycles are serviced at some settable ratio to correspond to their relative rates. Messages sent between switches are serviced in timestamp order. Since we are simulating mostly determinate programs with an invariable number of noncommunication operations and each is of a rather uniform duration, we can compute speedup "on the fly," as

$$\frac{\text{total time of essential operations}}{\text{simulated time}}$$

Unlike simulated time, essential operations do not include communication delay, so we are rightfully "charging" for them. Otherwise the measured speedup would be inflated.

Our simulator also makes it possible to measure the concurrency for infinitely many processors with no communication overhead. The simulator defines concurrency by using the notion of "ply" as follows: In ply number one, there is one packet, which corresponds to the operation initially demanded. Given ply n, ply $n+1$ is the union of all packets spawned by packets in ply n. By definition, all packets in a ply can be executed concurrently. The concurrency of a ply is therefore defined as the number of packets in it. The measured average concurrency gives a rough upper bound on the attainable speedup of a particular application. It thus provides a necessary condition for successful utilization of multiple processors. The Xputer network used in the present simulation is the rectangular *grid*. For a small number of nodes, as many as one hundred, this interconnection scheme should be adequate, since the worst-case delay of sqrt(n) is not appreciably different from log(n) for a minimal-delay configuration. Likewise, the grid configuration would probably be used if a subset of nodes were to be implemented on a single silicon wafer, although the interconnection of wafers might assume a different configuration.

Benchmarks. We have been running two kinds of benchmarks. One consists of toy programs which exhibit a single kind of activity, such as relatively independent process or pure divide-and-conquer functions. In part, they have been used to tune the parameters and load-balancing mechanism and demonstrate the system's effectiveness. For example, a fair amount of effort was required to bring the network to a point where n Xputers would spread a program evenly over n independent processes. Similarly, if there are, say, $2n$ independent processes being sequentially spawned, the network will run the first n until one completes, then fill the vacancy created with the $(n+1)$th, etc.

One example of a toy program was the divide-and-conquer factorial program (see p. 72). The upper curve in Figure 12 demonstrates the speedup for the computation of factorial 2^{10} on varying numbers of Xputers with square grid configurations.

The other class of benchmarks consists of more realistic applications that combine a number of activities in the areas of simple database searching and updating transactions and correlative signal processing. The lower curve in Figure 12 indicates speedups from one of these applications, a signal-processing problem that entails a moving-window correlation of two complex-valued streams of data. The computation for a single window consists of a weighted inner product of the current n values of one signal with the complex conjugate of the current n values of another. In the run shown, $n=20$ was specified, and 50 windows were computed concurrently. Runs with other parameters had a similar behavior.

Simulations also help quantify the locality effect by computing the distribution of inter-Xputer message dis-

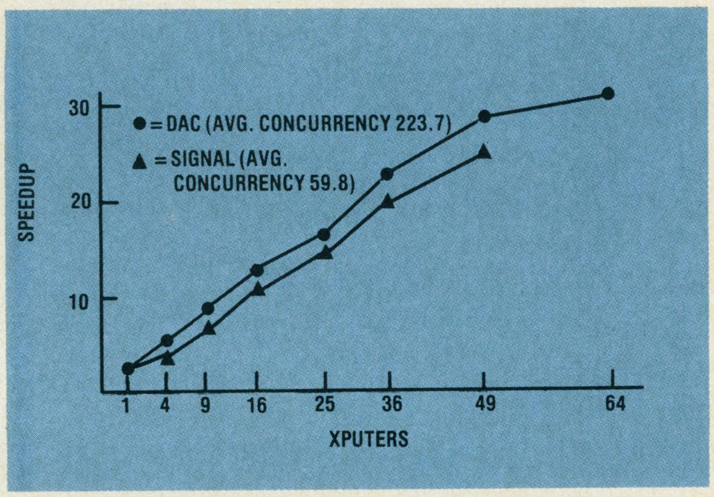


Figure 12. Speedup in two applications programs.

tances for data and APPLY packets. Figure 13 illustrates the average distances for the factorial and signal processing examples. We assume 10-μs-per-processor reduction operation and 40M-bps switch throughput, roughly what would be provided by a microprocessor with a customized switch. The indication of locality in the above examples is that the average distance a packet travels is considerably less than the worst-case distance. Another measured indication produced by the distinction between migrable and local functions is that typically only five to 20 percent of data references are nonlocal to an Xputer.

Future work

Due to resource limitations, we have not been able to consider all problems thoroughly but have shifted our attention from the evaluation model to load-distribution techniques, performance measurement, and language issues. We are preparing further optimizations of the evaluation model—in particular, the consumption of memory by the current graph-reduction evaluator. Simulator memory has limited the size of problems we have been able to explore. We thus believe the current Rediflow evaluator, which is an interpreter, can compete with a sequential processor only if the degree of concurrency in the application is very high. Some other good possibilities for improvement include replacing the interpreter with compiled native code.

In addition to continuing our evaluation and improvement of the basic Rediflow system, we are widening the investigation of application areas. For example, there is ongoing work on logic programs evaluation.[36,37] We also intend to engage in reliability studies.

An added feature of the mathematical model underlying functional evaluation is that data is never destroyed until its inaccessibility is established, which suggests that such a model might be a natural candidate for expressing the mechanization of recovery.[12,38] This possibility and our contention that the preferred configuration supports graceful degradation make Rediflow an attractive candidate for a reliability investigation.

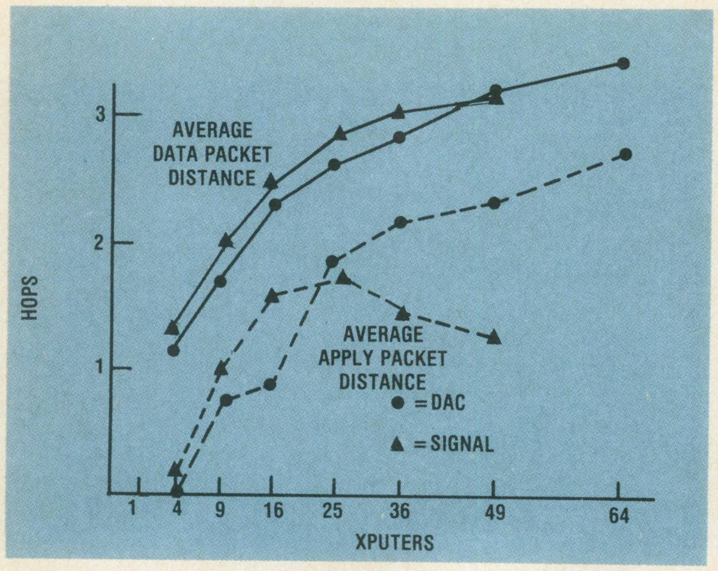


Figure 13. Packet transmission distances in two applications.

Related work

Rediflow is a outgrowth of earlier work on the Applicative Multiprocessing System at the University of Utah.[20] It differs from AMPS in its topology and its approach to load-balancing. Rediflow rejects the hierarchical approach in favor of a scheme with the potential for useful work in every physical node of the system. The retention of von Neumann processes is also new and reflects our belief that, for subprograms with a strong sequential orientation, the von Neumann architecture is still the fastest execution model. Finally, the Rediflow storage allocation and reclamation techniques are quite different from those of AMPS.

A number of other researchers have pursued goals similar to those of Rediflow. The Alice proposal,[39] for instance, is probably the closest in its evaluation model. It uses graph reduction, but since no distinction is made regarding fine-grain local operations and coarser grain migrable ones, it does not exploit these locality-enhancing aspects. A second difference is in the mechanics of node linkage during graph reduction.[20] Certain aspects of Hewitt's Apiary network[40] also seem similar in goals and mechanization to those of Rediflow. Rediflow also shares the use of pointers for establishing system-wide data structures with the much finer grained MIT "connection" machine.[41]

Dataflow machines also present comparable efforts.[16,17,42] They attempt to exploit concurrency of a finer granularity than Rediflow. Two observations are in order:

(1) The packets within a Rediflow Xputer can be processed concurrently or in an order-independent fashion. Dataflow architecture can likely enhance the performance *within* an Xputer node. Conversely, the load-balancing concepts we have outlined might provide an effective second level for managing work in a dataflow-oriented system.

(2) With the need to handle data structures, several lines of dataflow research seem to recognize that reduction models and demand-driven execution have something valuable to offer.[43,44] The reduction concept also underlies the conception of *I*-structures,[45] which are essentially demand-driven tuples or arrays.[46]

Among the language-oriented efforts, there are the Prolog-related approaches, such as Shapiro's Bagel.[47] The Concurrent Prolog language,[48] suggested for Bagel has roughly the capability of a functional language with unification pattern matching and indeterminate primitives in the form of guarded expressions.[49] Shapiro has suggested that programmers should explicitly specify the sites on which functions are executed in systolic-array fashion. Such specifications are an option to Rediflow, but we are encouraged by simulation results that do not require them. Other proposals intended to support full Prolog are being tested.[50] Since they involve an even more complex sequential model, it remains to be seen whether they will be effective for a wide range of problems or whether they will be relegated to a "back-end" role in a more general-purpose processor.

Finally, work is being done on general-purpose multiprocessors that are not language-driven.[51] Their performance-evaluation results add significantly to the information base and should provide valuable benchmarks for comparison against the more language-driven approaches, which are intended, after all, to simplify the use of multiprocessors.

The Rediflow multiprocessor system employs a packet-switching network to implement higher level programming abstractions. The intended applications exploit medium-grained or function-level concurrency and permit load-balancing in what is essentially a distributed system. Preliminary performance results show that the Rediflow architectural approach is promising. ✱

Acknowledgments

We wish to thank our colleagues at Utah—Gary Lindstrom, Elliott Organick, Tom Henderson, P.A. Subrahmanyam, and Richard Fujimoto—for their comments and continued encouragement. We also thank Peter Benson and Jiro Tanaka for programming assistance. Joe Fasel, Allan Gottlieb, Bert Halstead, Carl Hewitt, Paul Hudak, Dick Kieburtz, and Ronan Sleep made technical comments that contributed to our work. Finally, we thank Doug deGroot, Jan Galkowski, and Peter Kogge at IBM for their support and confidence.

This work has been supported at the University of Utah by grants from the IBM Corporation, National Science Foundation (MCS-8106177), and Defense Advanced Research Projects Agency of the US Department of Defense (Contract No. MDA903-81-C-0414), and during Keller's leave, at Lawrence Livermore National Laboratory by the US Department of Energy (Contract No. W-7405-ENG-48).

References

1. D. E. Muller and W. S. Bartky, "A Theory of Asynchronous Circuits," *Proc. Int'l Symp. Theory Switching,* 1959, pp. 204-243.
2. R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math,* Vol. 14, No. 6, Nov. 1966, pp. 1390-1411.
3. R. M. Karp and R. E. Miller, "Parallel Program Schemata," *J. Computing and Systems Sciences,* Vol. 3, No. 2, May 1969, pp. 147-195.
4. D. A. Adams, *A Computation Model With Data Flow sequencing,* Stanford University, Computer Science Dept., tech. report CS117, 1968.
5. S. Patil, "Closure Properties of Interconnections of Determinate Systems," *Proc. Project MAC Conf. Concurrent Systems and Parallel Computation,* June 1970, pp. 107-116.
6. R. M. Keller, "A Fundamental Theorem of Asynchronous Parallel Computation," T-y. Feng, ed., *Parallel Processing,* Springer-Verlag, New York, 1975, pp. 102-112.
7. W. H. Burge, *Recursive Programming Techniques,* Addison-Wesley, Reading, Mass., 1975.
8. P. Henderson, *Functional Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1980.
9. D. A. Turner, "The Semantic Elegance of Applicative Languages," *Functional Programming Languages and Computer Architecture,* Oct. 1981, pp. 85-93.
10. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing,* IFIP, North Holland, 1974, pp. 471-475.
11. R. M. Keller, "Denotational Models for Parallel Programs with Indeterminate Operators," *Formal Description of Programming Language Concepts,* E. Newhold, ed., Elsevier, North-Holland, 1978, pp. 337-366.
12. R. M. Keller and G. Lindstrom, *Toward Function-Based Distributed Database Systems,* University of Utah Computer Science Dept., tech. report UUCS-82-100, Jan. 1982.
13. M. Conway, "A Multiprocessor System Design," *AFIPS Conference Proc.,* 1963, pp. 139-148.
14. G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys,* Vol. 15, No. 1, Mar. 1983, pp. 3-44.
15. D. M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 365-381.
16. J. B. Dennis, "Data Flow Supercomputers," *Computer,* Vol. 13, No. 11, Nov. 1980, pp. 48-56.
17. I. Watson and J. Gurd, "A Practical Data Flow Computer," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 51-57.
18. D. P. Friedman and D. S. Wise, "CONS Should Not Evaluate Its Arguments," Michaelson and Milner, eds., *Automata, Languages, and Programming,* Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 257-284.

SOFTWARE PROFESSIONALS NEW ENGLAND/NATIONAL OPPORTUNITIES

Are you thinking of career advancement? Are you concerned that the person who represents you be as professional and informed in their field as you are in yours? Let E.P. Reardon Associates put their 20 years of successful placement counselling to work for you. Our Counsellors are knowledgeable professionals, in daily contact with companies around New England and, through our N.P.C. network, across the U.S.

If the challenge is gone or your career path is blocked in your present position, we have requirements for experienced professionals in the following areas:

Operating Systems Design	Scientific Applications
Computer Architecture	Compiler Design
Interactive Graphics	Office Automation
Peripherals Interfaces	Artificial Intelligence
CAD/CAM	
Networking	

To learn more about these and other openings call or, if you prefer, forward your resume to either address listed below. All inquiries will be answered within 48 hours and will be treated in complete confidence.

P.O. Box 1038
Burlington, MA 01803
617-273-5964

P.O. Box 228
Dedham, MA 02026
617-329-2660

E.P. Reardon Associates

Clients are EOE. Members of NPC.

19. C. P. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus,* Oxford University, PhD thesis, 1971.

20. R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely-Coupled Applicative Multi-Processing System," *AFIPS Conf. Proc.* Vol. 24, June 1979, pp. 613-622.

21. A. L. Davis and R. M. Keller, "Dataflow Program Graphs," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 26-41.

22. Arvind and K. P. Gostelow, "The U-interpreter," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 42-49.

23. J. Tanaka, *Optimized Concurrent Execution of an Applicative Language,* University of Utah, PhD thesis, Mar. 1984.

24. R. Keller, F.C.H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proc. Compcon Spring 84,* Feb. 1984, pp. 410-417.

25. M. DuBois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Trans. Computers,* C-31, No. 11, Nov. 1982, pp. 1083-1099.

26. C. V. Ravishankar and J. R. Goodman, "Cache Implementation for Multiple Microprocessors," *Proc. Compcon Spring 83,* Mar. 1983, pp. 346-350.

27. G. C. Fox, "Concurrent Processing for Scientific Calculations," *Proc. Compcon Spring 84,* Feb. 1984, pp. 70-73.

28. E. G. Coffman and P. J. Denning, *Operating Systems Theory,* Prentice-Hall, Englewood Cliffs, N.J., 1973.

29. A. Gottlieb et al., "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers,* C-32, No. 2, Feb. 1983, pp. 175-189.

30. I. Baron et al., "Transputer Does 10 or More MIPS Even When Not Used in Parallel," *Electronics,* Nov. 1983, pp. 109-115.

31. F. W. Burton and M. R. Sleep, "Executing Functional Programs on a Virtual Tree of Processors," *Proc. ACM Symp. Functional Programming Languages and Computer Architecture,* Oct. 1981, pp. 187-195.

32. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys,* Vol. 14, No. 1, Mar. 1982, pp. 93-143.

33. R. M. Keller and G. Lindstrom, "Applications of a Feedback in Functional Programming," *Conf. Functional Languages and Computer Architecture,* Oct. 1981 pp. 123-130.

34. H. G. Baker, Jr., "List Processing in Real Time on a Serial Computer," *Comm. ACM,* Vol. 21, No. 4, Apr. 1978, pp. 280-293.

35. P. Hudak and R. M. Keller, "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems," *Proc. ACM Symp. Lisp and Functional Programming,* 1982, pp. 168-178.

36. G. Lindstrom and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. 1984 Int'l Symp. Logic Programming,* Feb. 1984, pp. 168-176.

37. U. S. Reddy, "Transforming Logic Programs into Functional Programs," *Proc. 1984 Int'l Symp. Logic Programming,* Feb. 1984, pp. 187-196.

38. F. C. H. Lin, "A Distributed Load Balancing Mechanism for Applicative Systems," Department of Computer Science, University of Utah, PhD thesis proposal, Dec. 1983.

39. J. Darlington and M. Reeve, "A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Symp. Functional Programming Languages and Computer Architecture,* Oct. 1981, pp. 65-77.

40. C. Hewitt and H. Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence," *Proc. Compcon Spring 84,* Feb. 1984, pp. 418-423.

41. W. D. Hillis, *The Connection Machine,* Massachusetts Institute of Technology AI Laboratory, tech. report 646, Sept. 1981.

42. K. P. Gostelow and R. E. Thomas, "Performance of a Simulated Dataflow Computer," *IEEE Trans. Computers,* C-29, No. 10, Oct. 1980, pp. 905-919.

43. E. A. Ashcroft, unpublished presentation on a Lucid machine, Lawrence Livermore National Laboratories, Oct. 1983.

44. I. Watson, unpublished paper on dataflow research. Lawrence Livermore National Laboratories, Oct. 1983.

45. Arvind and R. E. Thomas, *I-structures: An Efficient Data Type for Functional Languages,* MIT Laboratory for Computer Science, tech. report MIT-LCS-TM-178, Sept. 1980.

46. R. M. Keller, "Divide and CONCer: Data Structuring for Applicative Multiprocessing," *Proc. 1980 Lisp Conference,* Aug. 1980, pp. 196-202.

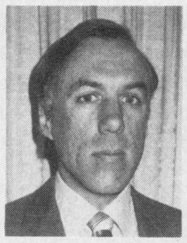
47. E. Y. Shapiro, Presentation on Bagel and Concurrent Prolog, *ACM Symp. Prin. Programming Languages,* Jan. 1984.

48. E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter,* Institute for New Generation Computer Technology, tech. report TR-003, Jan. 1983.

49. E. W. Dijkstra, "Guarded Commands, Non-Determinacy, and a Calculus for the Derivation of Programs," *Comm. ACM,* Vol. 18, No. 8, Aug. 1975, pp. 453-457.

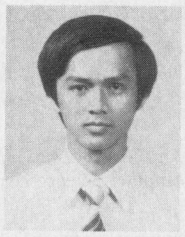
50. D. deGroot, ed., *IEEE Int'l Logic Programming Symp.,* 1984, entire publication.

51. E. F. Gehringer, A. K. Jones, and Z. Z. Segall, "The Cm* Testbed," *Computer,* Vol. 15, No. 10, Oct. 1982, pp. 40-49.



Robert M. Keller is a professor of computer science at the University of Utah. From 1970-1976 he was an assistant professor of electrical engineering at Princeton University. He has held visiting appointments at Stanford University and Lawrence Livermore National Laboratory. His primary interest is in highly concurrent and distributed models for evaluation of high-level language programs.

Keller received the MSEE from Washington University and the PhD from the University of California, Berkeley.



Frank C. H. Lin is currently working on his PhD degree in computer science at the University of Utah. He received a BSEE degree from the National Taiwan University in 1973 and a MSEE from Utah State University in 1978.

He was an engineer at Calcomp Electronics, Inc., 1976-1977. Since 1978 he has held various positions with the Computer System Division of Sperry Corporation in Salt Lake City, Utah. His research interests are computer architecture, networking, and applicative systems and fault-tolerant computing.

Questions about this article can be addressed to either author, Dept. of Computer Sciences, University of Utah, Salt Lake City, UT 84112.