2-1-1984

Rediflow Multiprocessing

Robert M. Keller
Harvey Mudd College

Frank C. H. Lin

Jiro Tanaka
University of Tsukuba

Rediflow Multiprocessing

Robert M.Keller
University of Utah
and
Lawrence Livermore National Laboratory

Frank C.H. Lin
University of Utah

Jiro Tanaka
University of Utah

Abstract

We discuss the concepts underlying **Rediflow**, a multiprocessing system being designed to support concurrent programming through a hybrid model of reduction, dataflow, and von Neumann processes. The techniques of automatic load-balancing in Rediflow are described in some detail.

1. Introduction

"Rediflow" is the name we give for a collection of ideas relating to multiprocessor system design and attendant software capabilities. The name is an elision of "reduction" and "dataflow", two models for evaluation of functional languages on multiple processors. As shall be seen, our conception also includes disciplined aspects of the "von Neumann" evaluation model as well.

1.1. Language and Software Issues

The motivation for use of functional languages stems from the fact that programs expressed in them usually contain a fair amount implicit concurrency, yet are **determinate**, or **speed-independent**, in that they are guaranteed to give the same results no matter how many processors are involved in their execution, and do so independently of the physical aspects of communication between those processors. As such, these languages seem to be ideal for the programming of multiprocessors when little concern over the distinctions between them and uniprocessors is desired. The determinacy criterion is essential for most applications, while intended exceptions can be handled with minor extensions to it. For example, we have successfully programmed distributed database applications, including those involving concurrent updating. Functional languages have other conceptual advantages, but space does not permit a lengthy discussion of them here. We claim that other types of languages, such as sequential languages and languages for logic programming, have an essentially functional character, and can be appropriately combined or embedded in them to share the advantages mentioned. One such combination is discussed herein.

This work has been supported by grants from the IBM corporation, National Science Foundation (MCS-8106177), Defense Advanced Research Projects Agency of the US Department of Defense (contract no. MDA903-81-C-0414) and, at LLNL, by U.S. Department of Energy (contract no. W-7405-ENG-48).

1.2. Hardware Issues

There presently seem to be no conceptual difficulties in interconnecting conglomerates of processors of arbitrarily-high peek processing capacities. Unfortunately, it is another matter to obtain useful work from such conglomerates. In order for a conglomerate to qualify as a **system**, it is necessary to provide linkages between the conception of problem solving and the hardware. Such techniques must be expressed in appropriate computer languages, which are then mapped for execution. Considerations of device technology being held invariant, it is the ease in performing this mapping which determines the relative 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 special purpose machine can be designed which will outperform all others on that application. Our goal here is not to address such machines, but rather to develop techniques which exploit multiprocessing power for a wide range of applications. A class of applications can be characterized by the **regularity**, **size span**, and **granularity** of its members.

Applications of high **regularity** contain many very similar operations which present similar computational demands. For these, approaches such as vector processors or cellular arrays may be most appropriate. The **static data flow** approach [6] also appears useful for applications of very high regularity, but less so for others, since it relies on a rather **balanced pipeline** approach to achieve speedup together with high utilization.

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 which can be contained in one array load, there may be extreme difficulties in "folding" larger problems to match the processor configuration. Even if such folding can be accomplished, it may result in significant unused processor cycles if not done with finesse.

Reprinted from *The Proceedings of COMPCON S'84*, 1984, pages 410-417.

The third differentiating characteristic mentioned is **granularity**. This term refers to the indecomposable units of work distributed to processors. Fine-grain operations would be on the level of bit operations, while slightly larger grains would be arithmetic operations. Large grains would be the level of processes or entire jobs. Rediflow is aimed at applications appropriate for **medium** (and larger) granularity, in which we intend to include irregularly-structured problems such as are found in, but not restricted to, the field of artificial intelligence. Other applications of medium granularity are certain adaptive numerical calculations, certain types of signal processing, and combinations of several application areas which interact in unpredictable ways.

2. Granularity considerations

Two areas of tradeoff which exist when considering granularity are communication overhead, and flexibility in load balancing. We first discuss communication. One reason why systems do not usually operate with peek speedup is that there must be some data communication between granules. This form of communication is minimal in the equivalent purely sequential computation. Therefore, attention must be given to reducing it in concurrent execution. For very small grains, the delay due to communication may exceed the delay of the operations themselves. For this reason, small granularity is not exploitable unless the regularity is so high that necessary communication paths are short and static, or unless there is little communication between grains. Widely distributing many small grains increases the likelihood that overhead due to communication will be large. Our conscious attempt at clustering small operations is one issue on which we seem to differ with other dataflow-related approaches (e.g. [28]). In summary, favoring large grains minimizes delay due to communication, but does so at the expense of loss of speedup due to concurrency.

As mentioned, 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 expenditure involving sending equal-size grains to all processing units. However, it will seldom be possible to make such determinations a priori. Instead, many applications will present work loads which are data dependent, and thus not susceptible to static analysis. To fully exploit the available multiprocessing resources, thus attaining maximum speedup, we need to have the ability to **dynamically** distribute load. Here we must pay attention to the tradeoff which favors small grains for the **ability** to balance more evenly, but which favors larger ones to minimize the total effort in actual distribution.

An area of concern often mentioned in relation to granularity is that of **context switching**, i.e. saving a processor's registers when it switches its attention from one unit of work to another, before the former unit is complete. This is therefore a technique for effectively reducing the grain-size, particularly if there is need to vary priority among large grains which may become temporarily inactive due to data dependencies (e.g. a process waiting for an i/o request to complete). As such, it seems fair to lump this overhead with that of load balancing.

3. System organization issues

In addition to intended application granularity, multiprocessing systems can be classified according to processor-memory structure. At one extreme, we have "dancehall" configurations, wherein one can imagine the system as having processors lined up along one side of a large dancehall, and memories along the other, with a network of switches in between. At the other extreme are "boudoir" configurations, in which each processor is closely paired with a memory, and a network of switches is used to communicate between such pairs.

Dancehall configurations appear to provide a uniform time access of any processor to any memory. However, this uniformity may disappear if there is significant contention at individual switches. Unfortunately, this delay also becomes uniformly longer with increasing numbers of processors and memories. It is possible to introduce caches which are coupled closely with processors and which retain local information for faster access, however this introduces the difficult problem of "coherence"(cf. [7, 24]): when one processor wishes to update information which has been cached by another, the latter must be invalidated, which entails additional communication overhead. Any machinery introduced to overcome this problem has a diluting effect on the useful capacity of the system. Boudoir configurations avoid this problem, since each processor has exclusive control over its own memory. This control also obviates introduction of special instructions for multiprocessor memory access, such as test-and-set and its derivatives [10].

4. Locality

The connection of a single processor with its memory has been pejoratively called the "von Neumann bottleneck" [1]. However, we are convinced that it is a powerful device, to be exploited as much as possible. A large number of such "bottlenecks" operating concurrently gives a very high aggregate bandwidth, much higher than a dancehall configuration with the same number of processors and memories, and with less attendant latency of memory accesses. Of course, these processor/memory pairs do not usually operate in isolation; however, if the communication between the components of the pair occur much more often than communication between pairs, in which case we say there is a high **locality**, then the boudoir configuration will be superior. It is conjectured that applications of medium grain and larger usually **do** possess sufficient locality to make the boudoir approach attractive.

When using large numbers (hundreds, to tens of thousands) of processors, it is not attractive to employ a centralized task queue from which processors seek work. One reason for this is that such a queue creates a bottleneck, and is contrary to reliability considerations. A

second, more subtle, reason is that such a queue tends to destroy locality, in that it homogenizes the distribution of data. As an alternative, we propose in Section 8 a method in which not only is the work distributed to the processors, but in which the method itself is also distributed.

5. Evaluation models

Several evaluation models have been suggested as the basis for multiprocessor execution. The most conventional of these entails extending the sequential von Neumann execution model to "processes" which run concurrently, but with various forms of communication between them. This method is a large-grain one, and has been most successful when processes are preassigned to physical processor-memory pairs [9]. Related, but much finer-grained are "dataflow" approaches, in which operations such as arithmetic are distributed to multiple function units and operands streamed through logical locations which feed such units. These have a potentially very high degrees of concurrency, but care must be taken, lest potential speedup be absorbed by communication overhead.

Another type of models is called "reduction", in which both the program and data are treated as an integrated, but distributed data structure. The spreading of this structure over the available processors permits its concurrent transmutation at many sites. A fine-grain string-reduction multiprocessor has been described by Mago [23]. Another string-reduction multiprocessor, of medium granularity, is presented in [20]. A medium-grain multiprocessor based on graph reduction is described in [14]. The approach of Rediflow is an extension of the latter.

In the reduction model of evaluation, no resident registers are employed, so cost of context switching is kept to a minimum, enabling rapid multiplexing of existing processor load in an effort to generate more load for concurrent execution. On the other hand, when the system is sufficiently loaded, such multiplexing should be abandoned in favor of more conventional sequential execution. One means of achieving this effect will be discussed later.

5.1. Evaluation by Graph Reduction

Our particular reduction evaluator can be derived from the lambda-calculus as a theoretical basis [4]. If one begins with a simple lambda calculus evaluator operating on string substitution, and introduces optimizations such as the use of pointers to sub-expressions rather than manipulating sub-expressions themselves, one is lead to a graphical, rather than string, representation. Attempts to make efficient the copying (which arise out of function applications, or equivalently "beta-reductions") inherent in this graph representation lead to the use of a linearized segment representation, in which the operator nodes of the graph correspond to words in the segment, and lists of addresses relative to the beginning of the segment represent arcs from the corresponding nodes. Values of "free" variables are imported in vectors, rather than employing an "association list" which must be searched repeatedly. More details on such representations may be found elsewhere [14, 5, 15]. It is worth noting that so-called **combinator** implementations [27] are also a form of fine-grained graph **reduction**.

A computation in this model begins as a single graph, with the result of one node "demanded" The demand then propagates to other nodes, some of which are primitive operators and others of which are defined by graphs of their own. The latter are expanded by virtually replacing the nodes with the defining graphs. A scheme similar to one described in [14] is employed for performing this virtual replacement through global address linkages.

As an example, suppose there is a tree-structured database distributed in the memory layer. This database may have been generated by some prior program, or explicitly loaded. Suppose further that we have a number of functions f1, f2, f3, ... each a "specialist" in performing a certain kind of search on the database. For example, one function might produce a certain "view" of the database, a sub-tree of nodes with a pre-specified property. A second might compute an aggregate function on the database, such as the number of positive nodes. A third might produce a transformed copy of the database, which is structurally the same, but having node values defined according to some mapping on individual nodes. All these functions could be performed concurrently on the database, each potentially recursively splitting into other function instances, perhaps creating its own data, which is made available for higher-level instances of the functions or for output. It is also possible for one function to be using another's output **while** the latter is being computed, rather than after it is computed. Such phenomena have all been demonstrated in Rediflow.

An advantage inherent in the reduction model is that **all** synchronization for the above activities is **implicit** in the underlying functional language implementation. This removes a considerable burden from the programmer. Whenever "strict" functions are involved (functions which require all of their arguments), the spawning of the necessary activities takes place automatically. This is a strong contrast to process-oriented models, in which there are three separate endeavors: setting up processes, synchronizing them, and using their values.

6. Integration of von Neumann Processes

Despite the advantages of the reduction model mentioned above, there remain aspects of applications which cannot exploit its inherent concurrency, synchronization, etc. It is not uncommon to find segments of applications which have a high peak concurrency, but have many internally **sequential** embedded segments. Such phenomena have been known since the earliest discussions of concurrent computation.

The effect of applying machinery powerful enough for concurrent computation to inherently sequential segments

is dilution of overall speedup. For example, a major difficulty with the reduction model is its memory intensiveness. It imitates an elegant mathematical model of functional languages, in which data values are never modified in place; they are only created, and destroyed (by storage reclamation). To do this for every conceivable operation means that much time is spent recycling storage. Although a certain amount of this can be done concurrently with other processing, the overhead seems to remain significant. A desirable goal is therefore to combine the load-spreading potential of reduction with other methods which are not so storage intensive. The approach taken in Rediflow entails what we call "von Neumann processes". These are encapsulated sequential processes which communicate with their environment in special ways. Externally, they appear as a form of "dataflow" functions, an observation made by Kahn [13]. Internally, von Neumann processes are ordinary sequential programs; operations appearing to be file input and output ("get" and "put") are used to communicate internal data values to and from the environment in the form of "tokens" moving on channels.

A key difference between our implementation of von Neumann processes and the suggestion of Kahn is that our implementation does not automatically supply **unbounded** buffers as channels. Instead, infinite buffers which can be attached to channels are naturally implemented in the reduction portion of the model, the operations of which are based on data structures rather than token passing. The interface from a von Neumann process to a reduction-implemented function solidifies a stream of token values into a stream data structure, while the interface from a reduction-function to a von Neumann process does the opposite. These functions are quite similar to ones which are used for external stream i/o in implementations of the reduction model [19].

Software networks of only one type of function can be connected together arbitrarily, the interface functions being used to connect networks of different types. The integration of the two models is done in such a way that what would have been merely **arcs** in distributed data structures can function as logical communication channels between processes. As such, the integration combines the "structure" and "token" models described in [5]into one unified system. Further details are given in [26].

As an example of the use of von Neumann processes, consider a function which performs the "APL-reduction" of a sequence (assumed non-empty) by a binary operator g (assumed non-associative), i.e. if the sequence is [x1, x2, ..., xn] the result is g[....g[g[x1, x2], x3], xn]. A pure reduction implementation would likely use an "accumulating" function such as (expressed in our language FEL [19])

```
reduce[g, x] =
  {
   result red1[head:x, tail:x]

   red1[accum, y] =
    if y =[]
        then accum
        else red1[g[accum, head:y], tail:y]
   }
```

Such an implementation would create n-1 instances of red1. This is inefficient, even with a built-in "tail recursion" optimization, since a von-Neumann process could evaluate the same function by the following sequential program:

```
reduce[g, x] =
  {*
   var y, accum;
   accum := head:x;
   y := tail:x;
   while y <> [] do
       begin
       accum := g[accum, head:y];
       y := tail:y
       end;
   return accum;
   *}
```

The above example does not demonstrate the use of channels. If the sequence were tokens from a channel, rather than components of a data structure, the corresponding von Neumann process might be

```
reduce[g, x] =
  {*
   var y, accum;
   accum := get:x;
   while more:x do
       accum := g[accum, get:x];
   return accum
   *}
```

Evaluation of the effectiveness of von Neumann processes is demonstrated in [26]. It should be noted, however, that they would not be superior if the operator g were associative, and data structures used for the sequence permitted easy concurrent decomposition (cf. [16]). In this case, a **divide and conquer** approach could be used, and for such, the reduction model seems well-suited.

7. Physical Configuration

As mentioned earlier, Rediflow currently assumes a configuration in which a number of processor-memory pairs are interconnected via a switching network. The combination of such a pair with an appropriate packet switch for information transfer will be called an **Xputer**, a primitive sketch of which is shown in Figure 7-1.

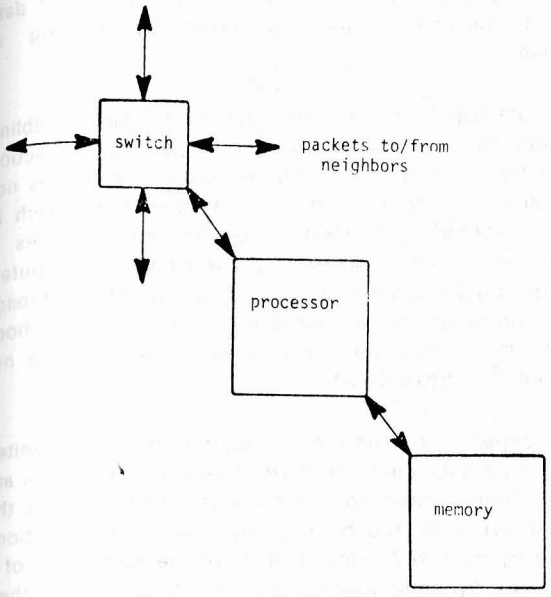

Figure 7-1: Sketch of an Xputer

The exact form of the Xputer network is not too important in this exposition. For a small number of nodes, say up to a few hundred, a rectangular **grid** interconnection should be adequate (see Figure 7-2). Input/output devices, which are not shown, may be attached at any nodes. For larger numbers of nodes, an interconnection topology with a lower worst-case delay is attractive. The concepts expressed in this paper can be used in such a system without modification.

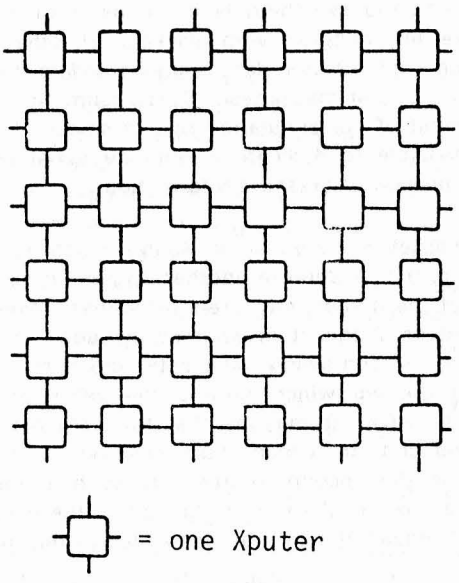

Figure 7-2: Sketch of an Xputer network

We can think of an Xputer grid as forming a plane surface, with the switches, processors, and memories each forming logically parallel **layers**. The layers need not be physically parallel. Interconnection exists only at the switch layer, while the memories in the memory layer have a combined **globally addressable** 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 then formed, which is then routed to the requesting Xputer. This request/return mechanism is integrated with the demand-drive mechanism of reduction evaluation, so that remote triggering of function evaluations can take place.

8. Load distribution and balancing

The refusal to rely upon a centralized queue for the distribution of work load means that other distribution methods must be used. As stated earlier, the smaller the grain, the more effective load balancing can be made. To avoid granularity so small that communication delays become significant, we aim at "medium" granularity. The approach taken in Rediflow is based on the contention that, ideally, grains behave as molecules of fluid being poured over a "surface" of processor-memory pairs. The reduction model enables this granularity, but we still need a means of making the fluid model work. We use the analogy of **pressure** to explain how this is done.

As with most multiprocessor organizations, queues are used to hold the backlog of work. In our case, the items on these queues are called **chares** (small tasks). Among several other queues to be described, each Xputer has a queue called the **apply queue** which is the reservoir of **migrable chares**. Chares on this queue represent function-instances which may be done on **any** available Xputer, due to the granularity and addressability assumptions stated earlier. Each such chare carries a "closure" which points to both a block representing the code of the function and a tuple of "imported" values, in addition to the actual argument, which may also be a tuple. Pure copies of such code blocks are cached locally in an Xputer, following an initial fetch from secondary or resident storage. There is no a priori correspondence between logical code and physical Xputer, and the same function may be executed in many different Xputers.

The number of chares on an Xputer's apply queue, weighted together with other resource utilization measures such as memory usage, can be thought of as defining its **internal pressure**. For the moment, assume that the Xputer can sense not only this pressure, but also the pressures of its neighbors, some function of which is called the **external pressure**. When the internal pressure sufficiently exceeds the external, some chares from the apply queue may issue forth into the interconnection network, where they are distributed to Xputers with lower pressures. In fact, Rediflow employs a moderately-intelligent switch, which is capable of directing chares along pressure **gradients** to find such low points. When a chare reaches an Xputer with a local pressure minimum, it is absorbed into its apply queue. This tends to raise the pressure of that Xputer, and lessen the likelihood that it will receive more chares, until its internal pressure becomes lower due to completion of work.

The phenomenon of **saturation** occurs when all Xputers are sufficiently busy that any attempt to migrate apply-chares would be futile, despite pressure differentials. An additional aspect of the Rediflow load balancing mechanism is the detection of such saturation. When external pressure is sufficiently high, migration attempts cease.

Obviously, pressure of Xputers is continually changing. Accordingly, it is necessary to continually update each Xputer's sense of its environmental pressure. This is done through a sampling process, in which the switch of each Xputer computes **transmitted pressure** as a function of transmitted pressures of its neighbors. One heuristic which seems to work well is to define the transmitted pressure to be 0 if the Xputer's internal pressure is below a certain threshold, and 1 + the **min**imum of the neighbor's transmitted pressures otherwise, with an absolute maximum on the order of the diameter of the network. This has the desired effect of permitting chares to flow toward the least loaded node.

9. Throttling

As mentioned earlier, an advantage of the reduction model of computation is that concurrently-executable work is easily spawned for migration to other processors. In effect, a "tree" is grown which corresponds to a single expression from which the "output" of the running program is extracted on a continuing basis. The default mode of servicing each Xputer's apply queue is FIFO, which traverses the tree **breadth first** and thus has the virtue of reaching concurrently executable nodes earlier. However, when saturation conditions exist, an Xputer switches to LIFO to give **depth first** traversal, in order to **throttle** its rate of chare production. This is helpful for avoiding queue overflows and for reducing the possibility of over-commitment of memory space, which could result in a kind of deadlock. (This suggestion was also made in [3].) In saturated mode, operators which would normally demand arguments concurrently are changed to demand them sequentially. This is easy to do within the reduction model. Finally, certain **eagerness** operators are normally compiled into the reduction code to cause anticipatory demands to components of suspended data structures [8] for added concurrency [17]. Eagerness operators are ignored in saturated mode.

10. Garbage collection

Distributing addressable memory across many modules, as is done in Rediflow, necessitates a distributed garbage collector. Although there are several candidates which suggest themselves, our current approach is to use a **copying** garbage collector [2]. In our distributed variation of this approach, the entire address space is divided in half, which appears as a halving of the memories in each Xputer. Allocation takes place within each Xputer from **successive** locations of its half-space. (Incidentally, the occupation level of this half-space contributes to the Xputer's internal pressure.) When all space is used up, accessible records are copied to the other half-spaces in the same Xputers. In this way, the distribution of data which is necessary for concurrent processing is maintained.

Distributed copying can be achieved by packets, enabling all Xputers to have an active role in garbage collection concurrently. (This packet-oriented implementation is not present in our current simulator). A related approach is used in Halstead's "Concert" [11], although he uses a "real-time" collector which tries to compact to one Xputer, rather than preserving the data distribution. Our approach can be converted to a real-time one, using methods similar to those described in [12], however we have not yet worked out these details.

Another aspect of garbage collection to be exploited concerns load balancing when von Neumann processes are involved. Due to their constantly regenerating nature, the latter are typically much larger-grained than functions evaluated by pure reduction, and therefore somewhat of a hindrance to dynamic load-balancing. Nonetheless, their contribution to Xputer pressure can be assessed by the presence of their components on internal queues, and they can be shifted from one Xputer to another with reasonable ease during garbage collection, when addresses are remapped anyway. The simulation of this form of balancing is currently not done.

11. General Packet Flow

A rough overview of the organization of an Xputer as explained above may be found in Figure 11-1. This diagram assumes that pressure sampling information is sent through the switching layer in the form of **packets**, which are intermingled with packets of other varieties (containing apply chares, data requests and responses, and garbage collection messages). This assumption has been used in most of our simulation results so far. However, it is also possible to dedicate a separate serial channel to the transmission of pressure information.

A **fetch** packet is issued by an Xputer which needs to get a datum from a location in another Xputer. It contains the address of the datum, and a return address. When a fetch packet arrives at the other Xputer's in-queue, the location is checked for containing valid data, and if so, a **forward** packet is created which returns the value to the first Xputer. However, it may be that the data have not yet been produced, in which case the return address is reserved in the second Xputer until such a time as the data are available. Also, if production of the data has not yet been demanded, it will be demanded at that time.

Because all result data have pre-allocated globally-addressable locations, it is not necessary to use any form of "token matching" [28] to get the data to their destinations. Thus, fast von Neumann-style memory is internally exploited in each Xputer. The use of addressing also permits routing tables to provide the shortest possible route to be chosen through the switching layer.

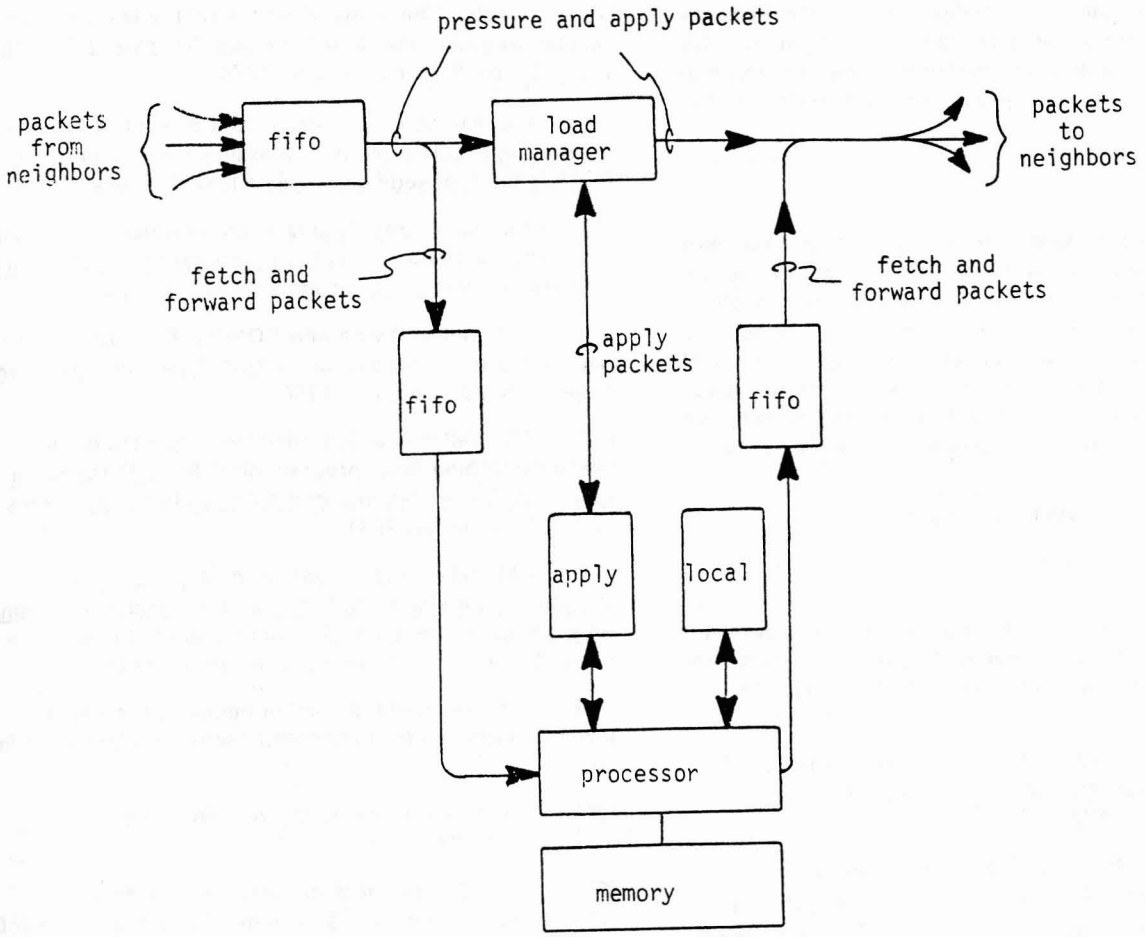

Figure 11-1: Packet flow within a Rediflow Xputer

12. Performance evaluation

The performance of the Rediflow architecture is being evaluated using simulation. As with most studies in their formative stages, we have begun evaluating speedups using an **introspective** model, i.e. one in which speedups are measured against a single processor with the same technological assumptions, architecture, and evaluation model as the multiprocessor. Due to certain needed improvements in our model, we are not yet ready to begin challenging existing sequential processors for applications with low degrees of concurrency. However, if the potential concurrency is high, then we believe Rediflow can exploit it with a demonstrated speedup.

We have been running two kinds of benchmarks. One consists of "toy" programs which exhibit a single kind of activity, such as pure "divide and conquer". The other consists of more "realistic" applications which combine a number of activities, in the areas of simple database searching and updating, and correlative signal processing. To briefly summarize, we have measured speedups in the range of 1 to 8 for the realistic applications, with fewer than 32 Xputers, and of up to 30 with the toy programs with as many as 128 Xputers. Memory space in our simulator is currently a principal limiting factor. In the process, we have demonstrated that the load distribution techniques designed for Rediflow apparently work well. They do exploit locality, in that typically over 50% of the data packets, and 80% of the apply packets, traverse paths of length at most 2. Usually fewer than 15% of all operations performed need to communicate outside one Xputer. We have also observed that the switches hypothesized for Rediflow do not seem to be a bottleneck under current technological assumptions.

13. Future Work

In addition to continuing our on-going evaluation and improvement of the basic Rediflow system, we are widening the investigation of application areas. For example, we and colleagues are in the process of including means of concurrently evaluating logic programs (cf. [22, 25]).

We also intend to engage in studies of reliability. An added feature of the mathematical model underlying functional evaluation is that data are never destroyed, making such a model a natural candidate for expressing a recovery model [18, 21]. This, coupled with our contention that physical configuration is apt to be more gracefully

degradable than a dancehall configuration, make Rediflow an attractive candidate for a reliability investigation. We hope to prove this, and other concepts discussed, through one or more physical multiprocessor realizations in the next few years.

14. Conclusions

We have presented a collection of ideas being integrated into a multiprocessing system called Rediflow, which employs a packet-switching network to implement higher-level programming abstractions aimed at efficiently running medium-grained applications with high degrees of concurrency. We have discussed a technique for load-balancing in an essentially-distributed system. Finally, we have explained preliminary results on performance of Rediflow.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8):613-641, August, 1978.

[2] H.G. Baker, Jr. List processing in real time on a serial computer. Communications of the ACM 21(4):280-293, April, 1978.

[3] F.W. Burton, M.R. Sleep. Executing functional programs on a virtual tree of processors. In Functional programming languages and computer architecture, pages 187-195. October, 1981.

[4] A. Church. The calculi of lambda-conversion. Princeton University Press, 1941.

[5] A.L. Davis and R.M. Keller. Dataflow program graphs. IEEE Computer 15(2):26-41, February, 1982.

[6] J.B. Dennis. Data flow supercomputers. IEEE Computer 13(11):48-56, November, 1980.

[7] M. DuBois and Faye A. Briggs. Effects of cache coherency in multiprocessors. IEEETC C-31(11):1083-1099, November, 1982.

[8] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In Michaelson and Milner (editors), Automata, Languages, and Programming, pages 257-284. Edinburgh University Press, 1976.

[9] E.F. Gehringer, A.K. Jones, and Z.Z. Segall. The Cm* testbed. Computer 15(10):40-49, October, 1982.

[10] A. Gottlieb, et al. The NYU Ultracomputer-Designing an MIMD shared memory parallel computer. IEEETC C-32(2):175-189, February, 1983.

[11] Robert Halstead. private communication, MIT, 1983.

[12] P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In Proc. Conf. on Lisp and Functional Programming, pages 168-178. ACM, ACM, August, 1982.

[13] G. Kahn. The semantics of a simple language for parallel programming. In Information Processing 74, pages 471-475. IFIPS, North Holland, 1974.

[14] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In AFIPS Conference Proceedings, pages 613-622. June, 1979.

[15] R.M. Keller and G. Lindstrom. Hierarchical analysis of a distributed evaluator. In Proc. International Conference on Parallel Processing, pages 299-310. August, 1980.

[16] R.M. Keller. Divide and CONCer: Data structuring for applicative multiprocessing. In Proc. 1980 Lisp Conference, pages 196-202. August, 1980.

[17] R.M. Keller and G. Lindstrom. Applications of feedback in functional programming. In Conference on functional languages and computer architecture, pages 123-130. October, 1981.

[18] R.M. Keller and G. Lindstrom. Approaching Distributed Database Implementations through Functional Programming Concepts. Technical Report, University of Utah, Department of Computer Science, 1982.

[19] R.M. Keller. FEL (Function Equation Language) Programmer's guide. 1982.AMPS Technical Memorandum No. 7.

[20] W.E. Kluge. Cooperating reduction machines. to appear in IEEETC , 1983.

[21] Frank C.H. Lin. A distributed load balancing mechanism for applicative systems. December, 1983.PhD Thesis Proposal, Department of Computer Science, University of Utah.

[22] Lindstrom, G. and Panangaden, P. Stream-Based Execution of Logic Programs. In Proc. 1984 Int'l. Symp. on Logic Programming. February, 1984. (to appear).

[23] G. A. Mago. A Network of Microprocessors to Execute Reduction Languages, Part I. International Journal of Computer and Information Sciences 8(5):349-385, March, 1979.

[24] C.V. Ravishankar and J.R. Goodman. Cache implementation for multiple microprocessors. In Compcon '83, pages 346-350. IEEE, March, 1983.

[25] U.S. Reddy. Transforming Logic Programs into Functional Programs. In Proc. 1984 Int'l. Symp. on Logic Programming. February, 1984. (to appear).

[26] J. Tanaka. Optimized concurrent execution of an applicative language. PhD thesis, University of Utah, Department of Computer Science, December, 1983.

[27] D.A. Turner. A new implementation technique for applicative languages. Software - Practice and Experience 9:31-49, 1979.

[28] I. Watson, J. Gurd. A practical data flow computer. IEEE Computer 15(2):51-57, February, 1982.