CYDRA™ 5

Directed Dataflow Architecture

Anderson
21 June

CYDROME™

SUMMARY

CYDROME™ has employed several new technologies in the Cydra 5:

- Directed Dataflow (Cydrome proprietary architecture)
- Compiler optimizing technology
- High-speed memory with guaranteed bandwidth
- Tightly integrated multiple optimized processors
- Parallel processing
- Parallelized UNIX® compatible operating system.

Cydrome has merged these innovative technologies into a balanced and functionally complete mini-supercomputer called the Cydra 5 Departmental Supercomputer. The Cydra 5 makes extensive use of industry standards, such as AT&T UNIX System V.3, ANSI FORTRAN 77, and the IEEE 754 floating point standard, to take advantage of customer's investments in applications software. In addition, the open architecture of the Cydra 5, including the use of the VME™ bus, makes it an attractive platform for systems integrators. Most important, the sophistication of the technology enables Cydrome to present customers with a very simple and familiar user interface.

INTRODUCTION

The building and testing of prototypes is as important to the engineer's job as the conducting of experiments is to the scientist's work. Also common to both activities are the high costs in time and money. Hence, computer simulations, which provide much faster turnaround and dramatically lower cost, have triggered a revolution in science and engineering.

The extremely visible success of computer simulations in such fields as computer-aided design has validated the use of this technique in engineering and science. As confidence in computer simulation has increased, so has dependence on it. Larger problems are being tackled and at a finer level of detail. The net result is a sharp increase in the need for computing power. Users who were formerly content with their departmental minicomputer now find it quite inadequate.

The architecture of the Cydra 5 Numeric Processor and the compiler technology that goes hand-in-hand with it, as well as the Cydra 5 system architecture, provide a complete solution to user needs. The chief virtue of this combination is its ability to excel with a broad spectrum of computations. It enables users to achieve large performance gains over superminis without re-engineering their existing application software and algorithms.

ARCHITECTURAL ALTERNATIVES FOR HIGH PERFORMANCE

The Nature of Parallelism in Applications

Any serious attempt at high performance computing involves the concurrent execution of multiple operations. The methods of achieving concurrent execution can be classified as fine-grained parallelism and coarse-grained parallelism.

Fine-grained parallelism is the simultaneous execution of multiple primitive operations such as additions and multiplications. This type of parallelism is usually exploited at a low level of operations in a piece of straight-line code or across the multiple iterations of an innermost loop. It is the form of parallelism used by uniprocessor architectures, such as vector and sequential processors, which overlap the execution of successive operations.

Coarse-grained parallelism refers to larger computations run in parallel and can be exploited by running multiple outer loop iterations or subroutine invocations in parallel on different processors. The term "parallel processing" generally refers to this type of computation. Each coars^-grained computation has some fine-grained parallelism that can be exploited by the individual processors in the parallel processor system. Thus, fine-grained and coarse-grained parallelism are complementary forms of parallelism that can be exploited individually or jointly.

Because most scientific and engineering programs are written in FORTRAN, a sequential language, it is up to the compiler to detect parallelism in the sequentially expressed program. The compiler must, in effect, prove to itself that two operations are independent of each other before it can exploit the parallelism inherent in a program.

Current compiler technology and the nature of the FORTRAN language allow the compiler to do a reasonably good job at the level of fine-grained parallelism. The task of proving that coarse-grained parallelism exists is considerably more difficult, if not impossible, for the compiler. As a rule, the user must either modify the program and indicate where opportunities for coarse-grained parallelism lie or use a language other than FORTRAN. Either alternative would simplify the compiler's task, but users are generally opposed to both alternatives.

The Attraction of Parallel Processing

Despite the difficulties, coarse-grained parallel processing continues to attract a lot of attention as a means of achieving high performance. The reason may be that the uniprocessor's performance does not increase proportionately with its price. Empirically, the achieved performance of the most cost-effective uniprocessor computer in each price band is proportional to the price of the product raised to a power less than 1. This is shown by the dashed curve in Figure 1.

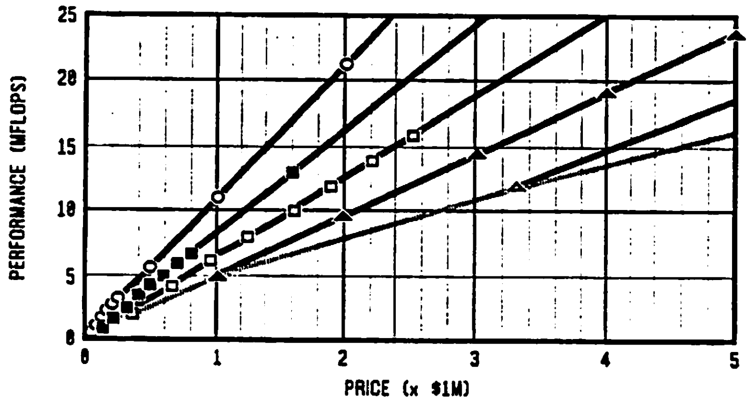
Assuming that on a single parallel job one could get a linear improvement in performance as the number of processors increased, one could achieve an arbitrarily large performance advantage over the best uniprocessor. This is demonstrated by the solid curves in Figure 1. Note that in this figure the generous assumption is made that the price of the parallel processor increases only linearly with the number of uniprocessors that comprise it. Furthermore, it would appear to be more advantageous to use many slower uniprocessors than a few faster processors. Therein lies the seductive appeal of the massively parallel processor.

In reality, the situation is quite different. As the number of processors increases, the time spent in executing the parallel portion of the job decreases but there is no effect on the sequential portion of the job. Eventually this fact determines the minimum execution time and limits the attainable performance, regardless of the number of processors used.

To make matters worse, executing a job in parallel invariably incurs some overhead cost. This overhead results from the time required to start up each parallel process, the time spent in communicating data from one process to another, and the time lost while one process waits on another or contends for some essential resource. The net effect is that the achieved performance eventually decreases as the number of processors increases. Each processor begins to spend more time on overhead activities than it does on useful work.

Figure 2 demonstrates this effect. Given a uniprocessor with a certain price and performance (lying on the dashed curve), the initial effect of increasing the number of processors is to increase the performance of the parallel processor above that of the equivalently priced uniprocessor. Beyond a certain point, however, the parallel processor's performance flattens and drops off, while the uniprocessor's performance rises steadily.

The proponents of parallel processing tend to focus on the speedup that is possible with multiple processors while ignoring the fact that an equivalently priced uniprocessor might give almost the same, if not better, performance. This is not to say that the parallel



Curves:

▬▬	Uniprocessor performance vs. price
-○-○-	Parallel processors composed of uniprocessors with a certain price and performance. The uniprocessor price and performance is indicated by the point at which the solid curve emanates from the dashed (uniprocessor) curve. Each solid curve shows the price and performance for 1 through 8 processors as well as for 16, 32, and 64 processors.
-■-■-	
-□-□-	
-▲-▲-	
-◆-◆-	

Figure 1. Comparison of Uniprocessor Performance With Idealized Performance of Parallel Processors

processor architecture is without merit. A parallel architecture makes sense in applications where the fraction of sequential computation and the overhead are small, or where the level of performance required makes it impossible to use a uniprocessor regardless of cost. Also, a parallel processor can be profitably used to run multiple independent jobs requiring a single processor for a job. This is the most common and beneficial way to use a multiple processor system.

In any event, whether a uniprocessor is intended for use as an individual processor or as the building block for a parallel processor, it is essential that its architecture be most effective at exploiting fine-grained parallelism.

Dataflow Architecture

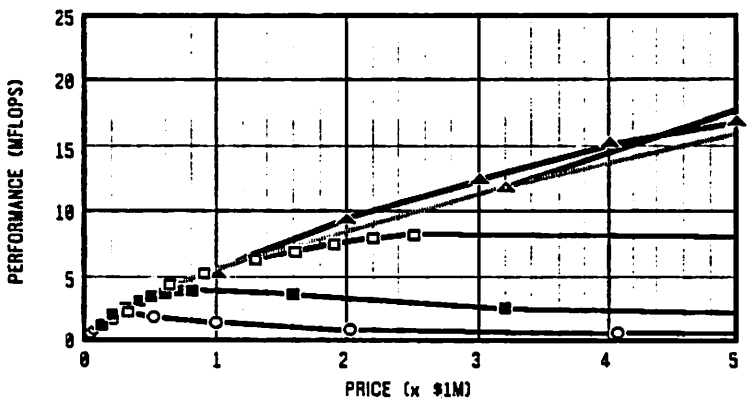
From a theoretical viewpoint, the most desirable architecture for fine-grained parallelism is dataflow. Dataflow is the only architecture that can exploit all forms of parallelism in a program; hence, it achieves higher performance over a broader class of computations than any other architecture.

In a dataflow processor, computation is viewed as a computation graph that explicitly represents *all* the dependencies between operations. Consider, for instance, the code segment and its corresponding computation graph in Figure 3. It is clear that the operation labeled A1 cannot be executed until operations M1 and M2 have completed, since M1 and M2 provide the inputs to A1. Operations R1, R2, R3 and R4, however, are all independent of one another and could be executed in parallel.

Of all processor architectures, the dataflow architecture places the minimum constraints on when operations may be executed. A dataflow processor can execute an operation any time after all its inputs are available, i.e., when all inputs have been computed and have arrived at the point of execution. By maximizing the number of operations that are eligible for execution at any point in time, the dataflow architecture can fully exploit the parallelism in an algorithm.

Despite these advantages, the dataflow architecture has failed to become a commercial success because of the extremely high overhead incurred at run-time. For each operation executed, five issues must be addressed at run-time:

- Will the operation be executed at all?
- If so, when will it execute?
- On which processing element will it execute?
- Where are the input operands located?
- Where will the result be placed?



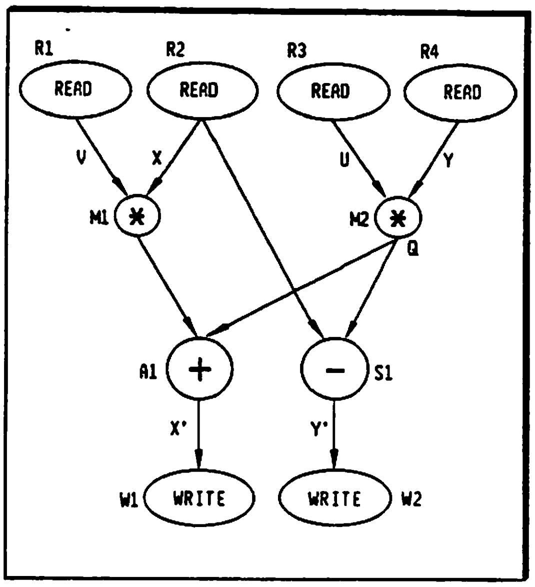
Curves:

▨ Uniprocessor performance vs. price

-○-○-, -■-■-, -□-□-, -▲-▲-, -△-△- } Parallel processors composed of uniprocessors with a certain price and performance. The uniprocessor price and performance is indicated by the point at which the solid curve emanates from the dashed (uniprocessor) curve. Each solid curve shows the price and performance for 1 through 8 processors as well as for 16, 32, and 64 processors.

Figure 2. Comparison of Uniprocessor Performance With the Realizable Performance of Parallel Processors

The net effect of this overhead is apparent when one considers that a dataflow machine with a peak performance of, say, 50 million floating-point operations per second would have to perform about 400 million associative searches per second--clearly an enormously expensive proposition.



$Q = U^*Y$

$Y = X - Q$

$X = Q + V^*X$

Figure 3. Code Segment and Corresponding Computation Graph

Directed Dataflow Architecture

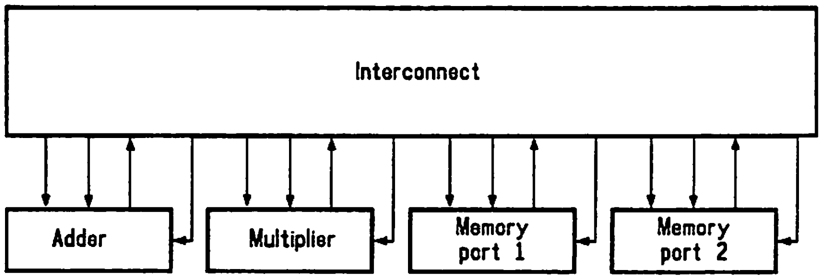
Directed Dataflow, the Cydra 5 proprietary architecture, retains the important benefits of the dataflow architecture but makes the concept commercially viable by moving as much decision-making as possible from run-time to compile-time. As a consequence, the cost of the hardware is comparable to that of other machines providing the same peak performance, but a significantly larger fraction of peak performance is consistently delivered.

In the dataflow architecture, the five issues listed above must be addressed at run-time for each operation that is executed. With the Directed Dataflow architecture, these issues are settled at compile-time to the extent possible. Where it is not possible, the processor hardware provides the support needed to resolve issues at run-time.

Scheduling of Computation Graphs. Conceptually, the compiler simulates the decision-making processes of the dataflow processor and creates a schedule that details when and where each operation is to be performed. The compiler has a slight advantage in that it can look ahead in the computation and make decisions that are globally more optimal. The dataflow processor cannot do this because its scheduling decisions are made in real-time.

The compiler's schedule is incorporated into the program that is executed by the Directed Dataflow processor. If the program specifies that a particular operation is to be executed at a particular time, one can safely assume that the inputs are available.

The actions of the compiler during scheduling are best illustrated by a series of simple examples. All the examples assume the simplified hypothetical processor shown in Figure 4. The processing elements (adder, multiplier, and two memory ports) are pipelined with the indicated latencies (the number of cycles to complete an individual operation), and each element is able to start a new operation every cycle. For simplicity, it is assumed that the interconnect can transmit results, in parallel, from the outputs of each processing element to either input of any processing element with no delay.



Pipeline stage time	1	1	1	1
Pipeline latency	2	3	5	5

Figure 4. Simplified Dircted Dataflow Processor

Scheduling of Straight-Line Code. Table 1 shows the schedule that the compiler would prepare for the code and the corresponding computation graph in Figure 3. The schedule is designed for execution on the processor in Figure 4.

Table 1. Compiler Schedule for the Code Segment in Figure 3

Time	Mem Port 1	Mem Port 2	Multiplier	Adder
0	R1	R2		
1	R3	R4		
2				
3				
4				
5			M1	
6			M2	
7				
8				
9				A1
10				S1
11	W1			
12	W2			

At the outset, the only operations that can be executed are read operations R1 through R4. Operations R1 and R2 are scheduled for execution at time 0 on Memory Ports 1 and 2, respectively. Given the 5-cycle execution latency for read operations, both inputs of M1 will be available at time 5 and may be scheduled for execution at 5 or any time thereafter. Operations R3 and R4 are scheduled for execution at time 1, making M2 eligible for execution at time 6 or any time thereafter.

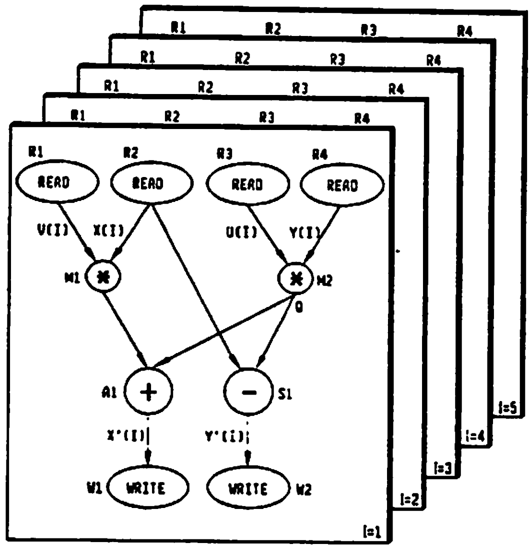
Operations M1 and M2 are scheduled for execution on the multiplier at times 5 and 6, respectively. Since the multiplier latency is 3 cycles, A1 and S1 are scheduled for execution at times 9 and 10, respectively. Write operations W1 and W2 have been delayed with respect to A1 and S1 by the 2-cycle adder latency and are scheduled at times 11 and 12, respectively.

This rather simple procedure can exploit the fine-grained parallelism within a single segment of straight-line code. The operations corresponding to the third FORTRAN statement, viz., R1, R2, A1, and W1, have been scheduled to execute in parallel with the operations of the first two FORTRAN statements. Assuming the same processing unit structure and pipeline latencies, a simple sequential processor would take at least 32 cycles to execute this code sequence, and even a rather sophisticated overlapped scalar processor would take 26 cycles. In contrast, the 13-cycle schedule for the Directed Dataflow processor represents a significant performance improvement, especially when one considers the simplicity of the hardware that controls the execution of the processing elements.

Nevertheless, the processor is considerably underutilized, owing to the pipeline latencies of the operations, the data dependencies between them, and insufficient amounts of parallelism. Note the number of empty slots in the schedule of Table 1. This processor, which is capable of starting four operations every cycle, ends up starting only 10 operations in 13 cycles, yielding a performance that is less than 25 percent of its peak performance.

Increasing the number of processing elements is not the solution. A processor with a larger number of processing elements would only achieve a lower utilization and very little improvement in performance. In fact, with an unlimited number of processing elements, the schedule length would reduce by only one cycle. As with parallel processing, it is far better to have fewer but faster processing elements. To use this processor more fully, one must exploit the far larger amounts of parallelism that exist between successive iterations of a loop.

Scheduling of Simple Loops. Let us now assume that the code sequence in Figure 3 is the body of an innermost loop, as shown in Figure 5. The computation graph now consists of multiple copies of the graph in Figure 3, with one copy for each iteration of the loop.
When there are no data dependencies between operations in different iterations of the loop, the dataflow architecture allows the processor to execute any number of iterations in parallel, limited only by the number of processing elements. For a compiler doing compile-time scheduling for the Directed Dataflow processor, the challenge is to exploit the inter-iteration parallelism to the point where the most heavily used processing element is fully utilized.



```
      DO 10 I = 1,N
      Q = U(I)*Y(I)
      Y(I) = X(I) -Q
      X(I) = Q + V(I)*X(I)
10    CONTINUE
```

Figure 5. Code for a Simple Vectorizable Loop and Computation Graph for the Loop Body

The first step is to determine which processing element is most heavily used. Each iteration performs two operations on the adder and multiplier, respectively, and six memory operations. Since there are two memory ports, this represents three operations per memory port. Thus, the memory ports are the most heavily used processing elements. The objective of maximizing performance on the loop can be served by scheduling successive iterations to start as frequently as possible. The interval between the initiation of two consecutive iterations is the "initiation interval". Clearly, the initiation interval cannot be less than the number of times the most heavily used processing element is used per iteration. (A shorter initiation interval would require the use of some processing elements more than 100%, which is impossible.) In our example, the minimum initiation interval is 3. This corresponds to optimal performance.

The schedule shown in Table 2 is based on the assumption that a new iteration is started every 3 cycles. Therefore, scheduling the first iteration implicitly schedules all subsequent iterations. For example, when R1 for the first iteration is scheduled on Memory Port 1 at time 0, corresponding R1 operations for subsequent iterations are implicitly scheduled at times 3, 6, 9, 12, et cetera. Thus, every time slot for Memory Port 1 which is at time 0 modulo 3 is crossed off as unavailable. With this additional "modulo" constraint, the scheduling of the rest of the operations proceeds as before.

The first time this constraint makes a difference to the schedule is when W2 is scheduled. From the viewpoint of input availability, W2 may be scheduled at time 12 or later. If it were scheduled at time 12, it would conflict with R2 of the fifth iteration. Thus, W2 is scheduled for execution at time 14, the next available time slot.

Table 2. Schedule for One Iteration of the Loop in Figure 5

Time	Time Modulo3	Mem Port 1	Mem Port 2	Multiplier	Adder
0	0	R1	R2		
1	1	R3	R4		
2	2				
3	0	---	---		
4	1	---	---		
5	2		M1		
6	0	---	M2		
7	1	---	---		
8	2			---	
9	0	---	---	---	A1
10	1	---	---	---	S1
11	2	W1		---	
12	0	---	---		---
13	1	---	---		---
14	2	---	W2	---	

This schedule can be replicated, with successive copies staggered at 3-cycle intervals. The iterations will dovetail perfectly. Table 3 shows the results when the schedule is executed at run-time. From the end of the initial start-up phase to the last few iterations, both memory ports--the most heavily used processing elements--are fully utilized. This result represents optimal performance. The processor now issues 10 operations every 3 cycles, which is 83% of its peak capability of four operations per cycle.

Table 3. Schedule of Multiple Iterations of the Loop in Fig. 6, Overlapped in Time

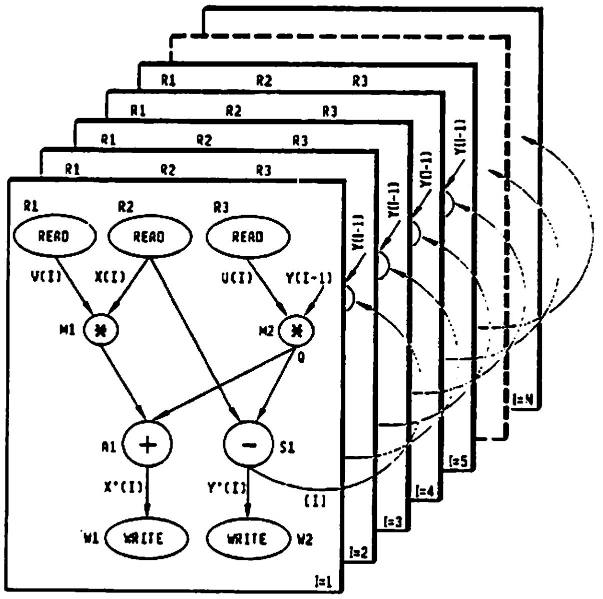
Time	Time Modulo3	Mem Port 1	Mem Port 2	Multiplier	Adder
0	0	R1	R2		
1	1	R3	R4		
2	2				
3	0	R1	R2		
4	1	R3	R4		
5	2			M1	
6	0	R1	R2	M2	
7	1	R3	R4		
8	2			M1	
9	0	R1	R2	M2	A1
10	1	R3	R4		S1
11	2	W1		M1	
12	0	R1	R2	M2	A1
13	1	R3	R4		S1
14	2	W1	W2	M1	
15	0	R1	R2	M2	A1
16	1	R3	R4		S1
17	2	W1	W2	M1	
18	0	R1	R2	M2	A1
19	1	R3	R4		S1
20	2	W1	W2	M1	
21	0	R1	R2	M2	A1
22	1	R3	R4		S1
:	:	:	:	:	:
:	:	:	:	:	:

Scheduling of Recurrence Loops. The loop in the previous example is one of the simpler types of loops; it contains no data dependencies between the operations in one iteration and the same operations in subsequent iterations. This type of loop can be "vectorized", i.e, reduced to a set of vector operations. All instances of M1, in all iterations, can be executed in their entirety as one vector operation; then all instances of M2 can be executed as one vector operation, and so on.

Now consider the example in Figure 6, where the value referenced as Y(I) on one iteration is the value computed for Y(I-1) on the previous iteration. In the computation graph of Figure 6, an arc is drawn from S1 in one iteration to M2 in the next iteration. This indicates that the result computed by S1 is used by M2 in the next iteration. This cyclic dependency is a recurrence; hence, this loop cannot be vectorized. Clearly, the vector operation corresponding to S1 cannot be performed until the vector operation for M2 is complete, because S1 is dependent on M2. Nor can the M2 vector operation be performed first, because the second operation in it is dependent on the first operation in the S1 vector operation. The sequence of M2 and S1 operations must happen in an interleaved order, which a vector processor cannot do. The vector processor would have to execute this loop in a degraded scalar mode.

Scheduling of recurrence loops poses no major problems for the Directed Dataflow processor. It only requires a different initiation interval from the one used in the previous loop schedule. The nature of the recurrence data dependency requires that M2 of the second iteration be scheduled at least 2 cycles after S1 of the first iteration. In the first iteration, S1 must be scheduled at least 3 cycles after M2. Therefore, the interval between the M2 operations for two consecutive iterations must be at least 5 cycles. After computing this initiation interval, the compiler constructs the schedule shown in Table 4.

The inter-iteration dependency prevents the compiler from overlapping successive iterations as much as processing element usage alone would have permitted. Although the Directed Dataflow processor's performance on the recurrence loop is 40 percent less than on the vectorized loop, it is considerably better than a vector processor could achieve using scalar execution.



```
      DO 10 I = 1,N
      Q = U(I)*Y(I-1)
      Y(I) = X(I) - Q
      X(I) = Q + V(I)*X(I)
10    CONTINUE
```

Figure 6. Code for a Recurrence Loop and Computation Graph for the Loop Body

Table 4. Schedule for One Iteration of the Loop

Time	Mem	Mem			
Time	Modulo5	Port 1	Port 2	Multiplier	Adder
0	0	R1	R2		
1	1	R3	R4		
2	2				
3	3				
4	4				
5	0	---	---	M1	
6	1	---	---	M2	
7	2				
8	3				
9	4				A1
10	0	---	---	---	S1
11	1	---	---	---	
12	2	W1	W2		

The advantage of the Directed Dataflow architecture becomes even more evident as the order of the recurrences increases. Suppose we change the first statement in the loop body (Figure 5) to read Y(I-2) instead of Y(I-1). The statement now constitutes a second-order recurrence, because the data dependency is between iterations that are two removed. Now the M2 operations from iterations that are two removed must be at least 5 cycles apart. In other words, twice the initiation interval must be at least 5 cycles; hence, the initiation interval must be at least 2.5, viz., 3. This initiation interval allows the Directed Dataflow processor to achieve the same performance it would achieve if the loop were vectorizable. It is this ability to perform well on linear as well as nonlinear recurrences that sets the Directed Dataflow architecture apart from the vector architecture.

Hardware Support for Loop Scheduling. A result generated by the execution of an operation must reside in some storage location, either memory or register, until is has been used by all the operations to which it is an input. Concurrent instances of a result must reside in different storage locations, and a mechanism must exist to ensure that each instance is matched to the correct operation. In the dataflow architecture, each iteration of a loop or invocation of a procedure constitutes a distinct "context" with a distinct name. Confusion is averted by tagging each result with the name of the context in which the result will be used. Associative searching is used to find matching inputs tagged with the same context name.

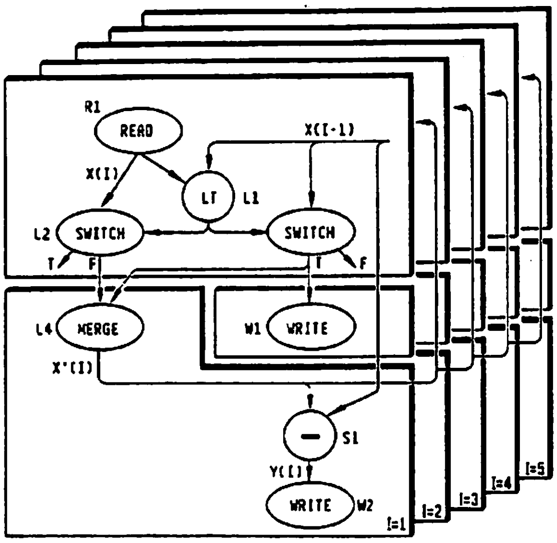
In the case of procedure calls, the conventional method of handling this problem is to allocate separate stack frames to hold the results computed on the various invocations of the procedure. While it appears that the results from each invocation are written to the same location, the frame pointer actually guides the results to equivalent locations in separate stack frames.

The Directed Dataflow architecture uses this conventional mechanism to handle multiple invocations of the same procedure. In addition, it employs iteration frames to handle parallel execution of multiple iterations of loops. At compile-time each result is assigned a definite location. At run-time the processor steers different instances of the same result to equivalent locations in separate iteration frames. This mechanism avoids the cost of associative storage.

Scheduling of Conditional Loop Bodies. One decision that cannot be made at compile-time is whether a data-dependent branch will be taken at run-time. Data-dependent branching complicates the task of loop scheduling, since the computations performed vary from one iteration to the next and make it impossible to devise a single schedule that can be replicated and overlapped at periodic intervals.

In the sequential mode of computing, the decision to execute an operation is made by data-dependent branches that direct the flow of control either to or away from the code containing the operation.

In the dataflow architecture, this issue is determined by data switches controlled by data-dependent conditions. The switches either inhibit or permit the input data to flow into the computation graph containing the operation. Consider the example in Figure 7 of a loop with some conditional branching. The switch operations transmit their input data down one of their two outgoing arcs (depending on the value of the boolean input) and control data flow into the subgraph of the computation. If the condition is false, no data flows into the subgraph consisting of operation W1. (This is the equivalent of branching away in the conventional sequential program.)



```
      DO 10 I = 1,N
      IF (X(I) .LT. X(I-1)X(I) = X(I-1)
      Y(I) = X(I) - X(I-1)
10    CONTINUE
```

Figure 7. Code for a Loop With Conditional Branching and Dataflow Graph for the Loop Body

Because data-dependent conditions obviously cannot be evaluated at compile-time, the Directed Dataflow architecture uses a different but equivalent approach. Instead of inhibiting the flow of data into a computation graph, it inhibits the *execution of the operations* in the graph. Control over the execution of operations is provided by a third boolean input to the operation. This input is computed at run-time and reflects the data-dependent condition. The boolean input is treated exactly like the other inputs for purposes of determining when an operation can be executed.
Figure 8 displays the Directed Dataflow computation graph for the example in Figure 7. Every operation now has an additional boolean input, shown entering the operation at the side. If the input is TRUE, the operation is executed normally. If it is FALSE, the operation becomes a "null" operation. (Operations shown without the boolean input actually have a boolean input that is constantly TRUE.) The SELECT operation selects either the left or the right input as its result, depending on the value of the boolean input. The hardware support for the conditional execution of operations makes it unnecessary to branch around the operations that will not be executed. Because no branching is involved in the Directed Dataflow computation, the task of scheduling the computation is no harder than in the earlier examples and proceeds as described in those examples.

Table 5 shows the resulting schedule. The initiation interval is 4 and is determined by the first-order recurrence involving the references to X(I) and X(I-1). In this example, the achieved performance, an iteration every 4 cycles, is better by a factor of at least 3 than the performance that would be achieved with traditional branching. This mechanism for handling loops with data-dependent branching applies to arbitrarily complex patterns of branching within the loop.

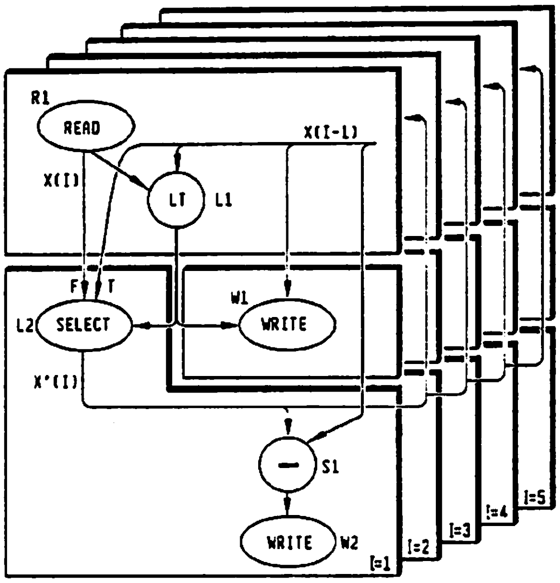


Figure 8. Directed Dataflow Computation Graph for the Loop Body in Figure 7

Table 5. Schedule for One Iteration of the Loop

Time	Time Modulo4	Memory Port 1	Memory Port 2	Multiplier	Adder
0	0	R1			
1	1				
2	2				
3	3				
4	0	---			
5	1				L1
6	2				
7	3	W1			L2
8	0	---			
9	1				---
10	2				S1
11	3	---			---
12	0	---			
13	1				---
14	2	W2			

Generality of the Directed Dataflow Architecture

Code containing fine-grained parallelism can be classified as:

- Code with vectorizable innermost loops
- Sequential code with little parallelism
- Code with innermost loops containing recurrences, condition branches, or irregular array accesses.

With vectorizable innermost loops, the Directed Dataflow architecture has a modest advantage over the vector architecture for two reasons. First, the Directed Dataflow architecture can chain an unlimited number of vector operations by spacing successive operations of one vector operation sufficiently to allow other vector operations to run concurrently, interleaved in time, on the same pipeline. This amortizes the vector startup penalty over a large number of vector operations. Second, because all the vector operations are chained, the Directed Dataflow architecture only needs to store a small part of each vector temporary; viz., the part that has been generated by one vector operation but has not yet been used as an input for the last time. Strip mining is unnecessary, which further reduces the vector startup penalty. (Strip mining is the partitioning of a long vector operation into short vectors the size of a vector register.)

With sequential code with little parallelism, the Directed Dataflow processor does better than the vector processor operating in its scalar mode. This is because the Directed Dataflow processor can execute operations out of sequence.

With code containing sets of innermost loops with recurrences, conditional branches, or irregular array accesses, the Directed Dataflow architecture has the greatest advantage. Whereas the vector processor drops to a sequential mode of execution, the Directed Dataflow processor continues to exploit any parallelism that exists.

Central to the superiority of the Directed Dataflow architecture are the compiler techniques and the supporting hardware. The hardware provides efficient allocation of register storage for the iteration frames and conditional execution of operations. The compiler and the hardware are inseparable design considerations. Even the most complex compiler techniques would fail to fully exploit fine-grained parallelism if the hardware failed to provide the appropriate architectural features.

THE CYDRA 5 SYSTEM

The Cydra 5 Departmental Supercomputer is a heterogeneous multi-processor system designed to be a functionally complete data processing solution for serious users in the engine·ring and scientific disciplines. It draws upon the most appropriate technology to meet each need.

As shown in Figure 9, the Cydra 5 is designed around a central bus and supports three types of processor: the Numeric Processor, the Interactive Processors (general processors), and the I/O Processors. Each of these processors is optimized for a particular type of task.

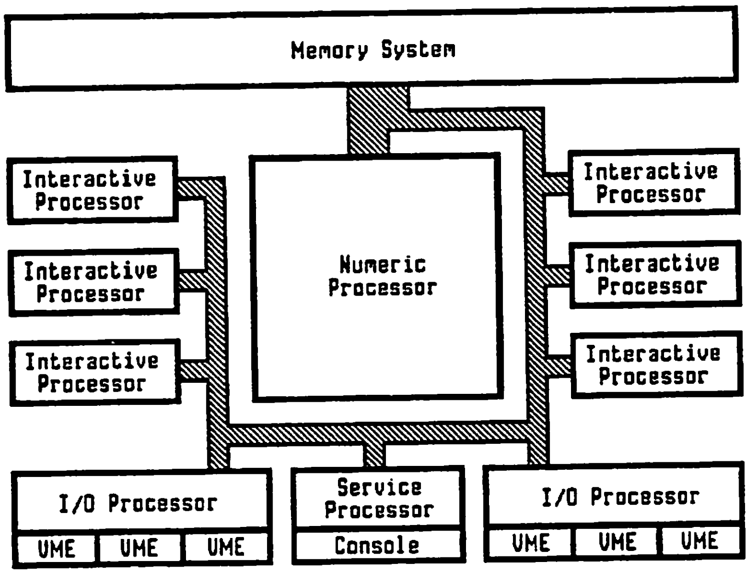


Figure 9. Cydra 5 System Diagram

The Numeric Processor has been optimized for the task of running large floating-point intensive applications. By virtue of its Directed Dataflow architecture, it can sustain high performance over a much broader spectrum of computations than other mini-supercomputer processors. Consequently, it is better equipped to meet the user's need for high performance without re-engineering of the application software.

Maintaining a balance with the high-performance Numeric Processor requires a high bandwidth memory system. Although the Numeric Processor has an instruction cache, it avoids data caching in order to avoid the anomalous performance that results when working with large data sets. The highly interleaved main memory incorporates a unique architecture that guarantees a uniformly high memory bandwidth regardless of how data is placed and referenced in memory. These features have been provided to meet a very important design objective: the user must be able to use the computer without perceiving any anomalous characteristics or performance shortfalls.

While the Numeric Processor is intended to execute numerically intensive applications, Cydrome recognizes that the typical user will run other jobs that are not numerically intensive, such as text editors, compilers, and interactive tasks. Rather than tie up the Numeric Processor with such tasks, Cydrome designed a tightly integrated general-purpose subsystem that shares memory with the Numeric Processor. This subsystem provides most of the operating system services, leaving the Numeric Processor free to run applications continuously. By tightly integrating the Numeric Processor and the general-purpose subsystem, Cydrome avoided the clumsiness of the host/attached-processor combination.

To avoid the bottleneck that occurs when the operating system does not execute the application's I/O requests fast enough, Cydrome has designed a UNIX V.3-compatible operating system with greatly improved I/O handling capability. This CYDRIX™ operating system executes on the general-purpose subsystem, which contains multiple general-purpose processors (the Interactive Processors). Cydrix is designed to execute as a set of parallel processes operating in a symmetric parallel mode on multiple processors.

In running Cydrix, the general-purpose subsystem functions as a parallel processor. UNIX, being a procedure-oriented system, is inherently parallelizable. Its execution consists of the joint activity of all user processes that are currently in kernel mode. This potential parallelism has been successfully exploited in Cydrix by modifying the kernel to make it re-entrant.

With multiple independent non-numeric user tasks, the Interactive Processors function as a multiprocessor, achieving near-linear speedup with the number of processors. In this mode, the Interactive Processors yield considerably better cost-performance than an equivalently priced uniprocessor.

High I/O performance is also achieved by using multiple processors. These microprocessor-based I/O processors can perform gather/scatter operations and handle tens of I/O transfers in both directions simultaneously. The total I/O bandwidth and storage capacity are more than adequate to ensure a balance with the performance capability of the Numeric Processor.

CONCLUSION

The Cydra 5 Departmental Supercomputer combines a radically different internal design with a familiar and comfortable user interface. It achieves significantly higher performance than superminis and higher cost-performance than conventional mini-supercomputers without re-engineering of applications. Cost-effective Directed Dataflow architecture and sophisticated compiler technology work hand-in-hand to provide an innovative solution to the computing needs of the engineering and scientific communities.

SUMMARY OF CYDRA™ 5 SPECIFICATIONS

Numeric Processor	Directed Dataflow™ architecture	
	Context Register Matrix	
	Conditional Scheduling Control	
	Floating Point Arithmetic: IEEE 754 Standard	
	Arithmetic precision: 32-bit and 64-bit	
	Instruction Cache: 32 kilobyte	
	Cycle Time: 40 nanoseconds	
Interactive Processors	General purpose 32-bit processors	
	Cache per processor: 16 kilobytes	
	Multiprocessor cache coherency hardware	
	Maximum configuration: 6 Interactive Processors	
Input/Output Processors	Sustained transfer rate per processor: 40 megabytes/second	
	Gather/Scatter data transfer	
	Industry-standard VME bus interface	
	Maximum configuration: 2 Input/Output Processors	
	Maximum number of simultaneous VME buses: 6	
	Maximum number of simultaneous I/O controllers: 30	
Memory Subsystem	Virtual Address Space: 4 gigabytes	
	Main Memory	Capacity: 8 megabytes to 256 megabytes
		Sustained transfer rate: 400 megabytes/second
		Stride insensitive design
		Up to 64-way interleaving
	Support Memory	Capacity: 8 megabytes to 64 megabytes
		Optimized for rapid data access
Peripheral Devices	Disk: 830 megabytes per drive with 2.5 megabytes/second transfer rate	
	Tape: 6250 bpi with 75 ips in start/stop mode	
	Printer: 600 lpm	
	Communications: RS-232C and Ethernet™ support	
Cydrix™ 5.3 Operating System	Compatible extension of AT&T® UNIX® System V.3	
	Transparent multiprocessing	
	Dynamic load balancing	
	Extent-based file system	
	Buffered and unbuffered input/output	
	Asynchronous input/output	
	Disk striping	
	Batch queue facility	
	TCP/IP support	
	Remote Graphics Library	
	UNIX tools and utilities	
	Application performance profiling tools	
	Socket library compatible with Berkeley 4.2 UNIX	