THE MANCHESTER PROTOTYPE DATAFLOW COMPUTER

The Manchester project has developed a powerful dataflow processor based on dynamic tagging. This processor is large enough to tackle realistic applications and exhibits impressive speedup for programs with sufficient parallelism.

J. R. GURD, C. C. KIRKHAM, and I. WATSON

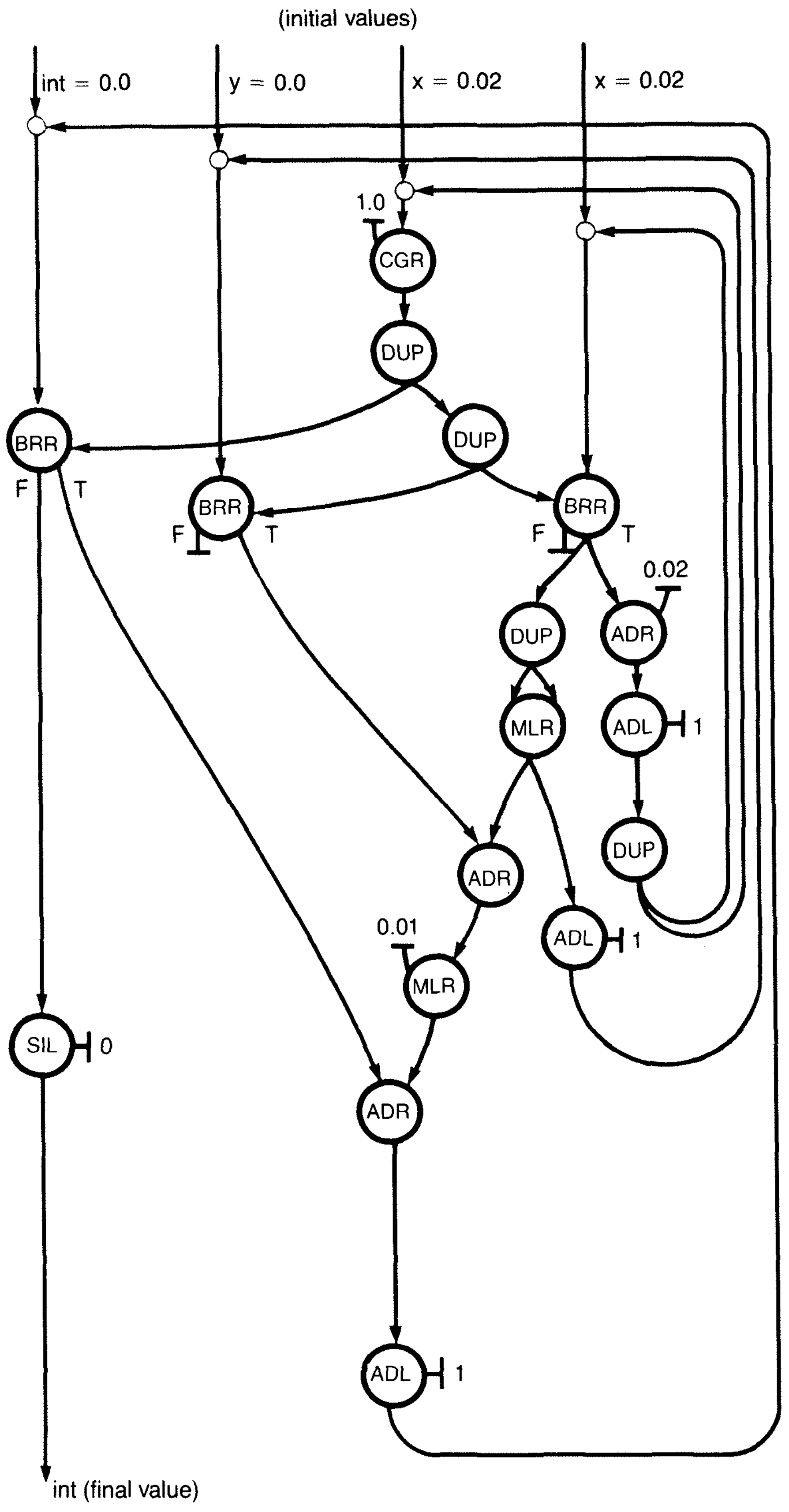
INTRODUCTION

Since about 1970 there has been a growing and widespread research interest in parallel data-driven computation and dataflow computer architecture. Centers of expertise in dataflow techniques have emerged at MIT in the United States, CERT-ONERA in France, NTT and ETL in Japan, and the authors' establishment in the United Kingdom. This interest has culminated in many designs for data-driven computer systems, several of which have been or are in the process of being implemented in hardware. For example, a machine based on the tagged-token model of dataflow computation has been operational at the University of Manchester since October 1981. This article reviews the architecture and performance of this machine.

Dataflow is a technique for specifying computations in a two-dimensional graphical form: Instructions that are available for concurrent execution are written alongside one another, and instructions that must be executed in sequence are written one under the other. Data dependencies between individual instructions are indicated by directed arcs, as shown for a small program in Figure 1. Instructions do not reference memory, since the data-dependence arcs allow data to be transmitted directly from generating instruction to subsequent instruction. Consequently, instructions can be viewed as pure operations—this perspective is described in the Dataflow Programs section. Each instruction can be activated independently by incoming data values: Execution commences as soon as all required input values for that instruction have arrived (as in the execution sequence of Figure 4).

Dataflow systems implement this abstract graphical model of computation. Individual systems differ mainly in the way they handle reentrant code. Static systems do not permit concurrent reactivation, and so they are restricted to implementing loops and cannot accommodate recursion. Dynamic systems permit recursive reactivation, either by code-copying or by tagging, at every occurrence of reentry. The nature of a system determines the types of language features that can be supported—recursion, for example, *cannot* be handled by static systems. The structure of a dataflow computer follows the model of message-passing multiprocessors. The Manchester project has designed a powerful dataflow processing engine based on dynamic tagging. The system is now running reasonably large user programs at maximum rates of between 1 and 2 MIPS (million instructions per second). Details on the architecture of this system are given in The Manchester Dataflow Processor section.

To date, few details have been published on the performance of operational dataflow hardware—after all, only a few of the larger systems have been active for longer than a year. Skepticism about the potential of dataflow techniques will persist until good performance figures can be demonstrated. First attempts have been made to define the objectives for performance evaluation for dataflow hardware, and some preliminary results from the Manchester prototype system are presented here. The strategy for evaluation is presented in the System Evaluation Strategy section, along with a discussion of program characteristics and their measurement on a dataflow simulator. The Benchmark Process section presents some details of the benchmark pro-



The final stage of translation forms a machine-code program with its input data. This two-dimensional graphical form is traditionally used to present dataflow programs. The nodes of the graph represent machine instructions, while the arcs represent data paths between instructions. It will be noticed that the branch (BRR) instructions behave as two-way switches inserted in the arcs, and that where a token is required as input to more than one instruction it has to be replicated using explicit duplicate (DUP) instructions.

FIGURE 1. Dataflow Graph for the Integration Program

grams that have been executed, and the Evaluation Results section presents the results obtained when these programs were executed on the prototype hardware.

Programs with large data structures have revealed that there is a need for hardware with specialized structure-storing capabilities. A structure-store unit is being designed to accommodate this need. In the long term, the use of multiple rings opens the possibility of incrementally expandable computing power in a dataflow multiprocessor. The prospects for such extensions to the existing system are discussed in the Future Directions section.

DATAFLOW PROGRAMS

Dataflow programs can be written at a high, an intermediate, or a low level. Figure 2 shows a program for computing the area under the curve $y = x^2$ between $x = 0.0$ and $x = 1.0$. It is written in the high-level single-assignment language SISAL, a typical Pascal-like dataflow language. SISAL's single-assignment property dictates that each variable be assigned only once in a program. This gives the language cleaner-than-usual semantics and makes it easier for the compiler to exploit program parallelism. Of course, parallelism could be extracted from programs written in more conventional languages, but the extraction process would be complex

```
export Integrate

function Integrate (returns real)

for initial
     int := 0.0;
     y   := 0.0;
     x   := 0.02
while
     x < 1.0
repeat
     int := 0.01 * (old y + y);
     y   := old x * old x;
     x   := old x + 0.02
returns
     value of sum int
end for

end function
```

Dataflow applications programs can be written in high-level programming languages in exactly the same way as for conventional computer systems. The most convenient type of language for compiling dataflow code is known as a single-assignment language. This type of language has a syntax similar to that of conventional languages like Pascal but has nonsequential semantics (i.e., it offers concurrent control constructs). An example program written in the single-assignment language SISAL is shown here. The program computes the area under the curve $y = x^2$ between $x = 0.0$ and $x = 1.0$ using a trapezoidal approximation with constant x intervals of 0.02.

FIGURE 2. Integration Program in the High-Level Programming Language SISAL

and would obscure important principles that are naturally apparent in SISAL.

Compilation of the high-level programs first translates the text into an intermediate-level (or compiler target) language roughly equivalent to a conventional macroassembler language. Figure 3 shows an abbreviated form of the intermediate code produced by the SISAL compiler for the program in Figure 2. Here, the template assembler language TASS is used. The main features of the translated program are that the variables (*int*, *y*, *x*, etc.) can be identified with the SISAL program text, whereas the operators (CGR, SIL, BRR, etc.) can be identified with the dataflow instruction set. The abbreviated form of Figure 3 is for the sake of clarity, because the "invented" variables would normally be given unintelligible names and a lot of redundant assembler code would be produced. In essence, Figure 3 shows the form of a program written directly at the intermediate level.

The final step of the compilation is to generate code and data files representing the machine-level program. Manchester machine code is relocatable via a segment table (see the next section) that identifies a base address and limiting offset for each of 64 code segments. Consequently, the code file contains segment table entries as well as the instruction store contents. Each instruction comprises an opcode and a destination address for the instruction output, together with an optional second destination address or a literal operand. The data file contains the initializing values, which represent the program input. Each entry consists of a typed data value and a three-field tag, together with the destination address to which the input should be sent.

Code at any level can be represented graphically, since statements specify paths to be followed by data passing between operators. In particular, it is traditional to represent the machine-level code as a directed graph. Figure 1 shows the machine code generated for the integration program in Figure 3.

The integration program is an example of a reentrant program—that is, one that reuses part of itself. Each separate iteration reuses the same code but with different data. To avoid any confusion of operands from the different iterations, each data value is *tagged* with a unique identifier known as the *iteration level* that indicates its specific iteration. Data are transmitted along the arcs in tagged packets known as *tokens*. Tokens for the same instruction match together and instigate the execution of that instruction only if their tags match.

The idea of tags can be extrapolated to encompass reentrant activation of complete procedures, thereby allowing concurrent executions of the same procedure to share one version of its instruction code. This is achieved by extending the tag with an *activation name*, which must also match. The activation name is also used to implement recursive functions, which need tags to generate a parallel environment analogous to the "stack" environment used in sequential language implementations.

```
(\I "TASS" "TSM");

!    Integration by trapezoidal rule
!    ===============================

!    initialize the loop variables

int       = (Data "R 0.0");
y         = (Data "R 0.0");
x         = (Data "R 0.02");

!    merge the initial values with the loop output values

int_mrg   = (Mer int new_int);
y_mrg     = (Mer y new_y);
x_mrg     = (Mer x new_x);

!    test for termination of loop

test      = (CGR "R 1.0" x_mrg);

!    gate the loop variables into new loop instance or direct result to output

gate_int = (BRR int_mrg test);
old_int  = gate_int.R;
old_y     = (BRR y_mrg test).R;
old_x     = (BRR x_mrg test).R;

result    = (SIL gate_int.L "O 0").L;

!    loop body : form new values for loop variables

incr_x    = (ADR old_x "R 0.02");
x_sq      = (MLR old_x old_x);
height_2 = (ADR old_y x_sq);
area      = (MLR "R 0.01" height_2);
cum_area = (ADR old_int area);

!               : increment iteration level for new loop variables

new_int   = (ADL cum_area "I 1").L;
new_y     = (ADL x_sq "I 1").L;
new_x     = (ADL incr_x "I 1").L;

!    output the final value of int

            (OPT result "G 0");

(Finish);
```

Programs written in SISAL are translated into an intermediate language such as TASS. Other high-level languages can be translated into this intermediate form, or programs may be written in TASS directly. For simplicity, the version of the integration program shown here is not a compiled version of the SISAL program in Figure 2, but an assembly-level program for the same task. However, the influence of the high-level version can be seen in the shape of this lower level program. The Manchester dataflow machine code is used in this figure. The Manchester system is an example of a "tagged-token" dataflow machine, which uses tag fields to distinguish reentrant activations of shared code. The "iteration-level" tag field is used to separate loop activations, using the ADL and SIL instructions. The effect of program "jumps" is achieved by the branch instructions. The remaining instructions are normal arithmetic/logic operations. The following mnemonics have been used:

ADR—add floating-point values
BRR—branch
CGR—compare floating point l.h. > r.h.
ADL—add to iteration level
MLR—multiply floating-point values
OPT—send output to host processor
SIL—set iteration level

FIGURE 3. Integration Program in the Template Assembly Language TASS

A final cause of reactivation is the reuse of code to process different parts of a data structure, for instance, an array. This is achieved by another extension to the tag, known as the *index*.

The above model of computation is known as tagged-token dataflow. It is the basic model implemented by the prototype Manchester hardware. Note, for example, the use of the tag-manipulating instructions ADL (add

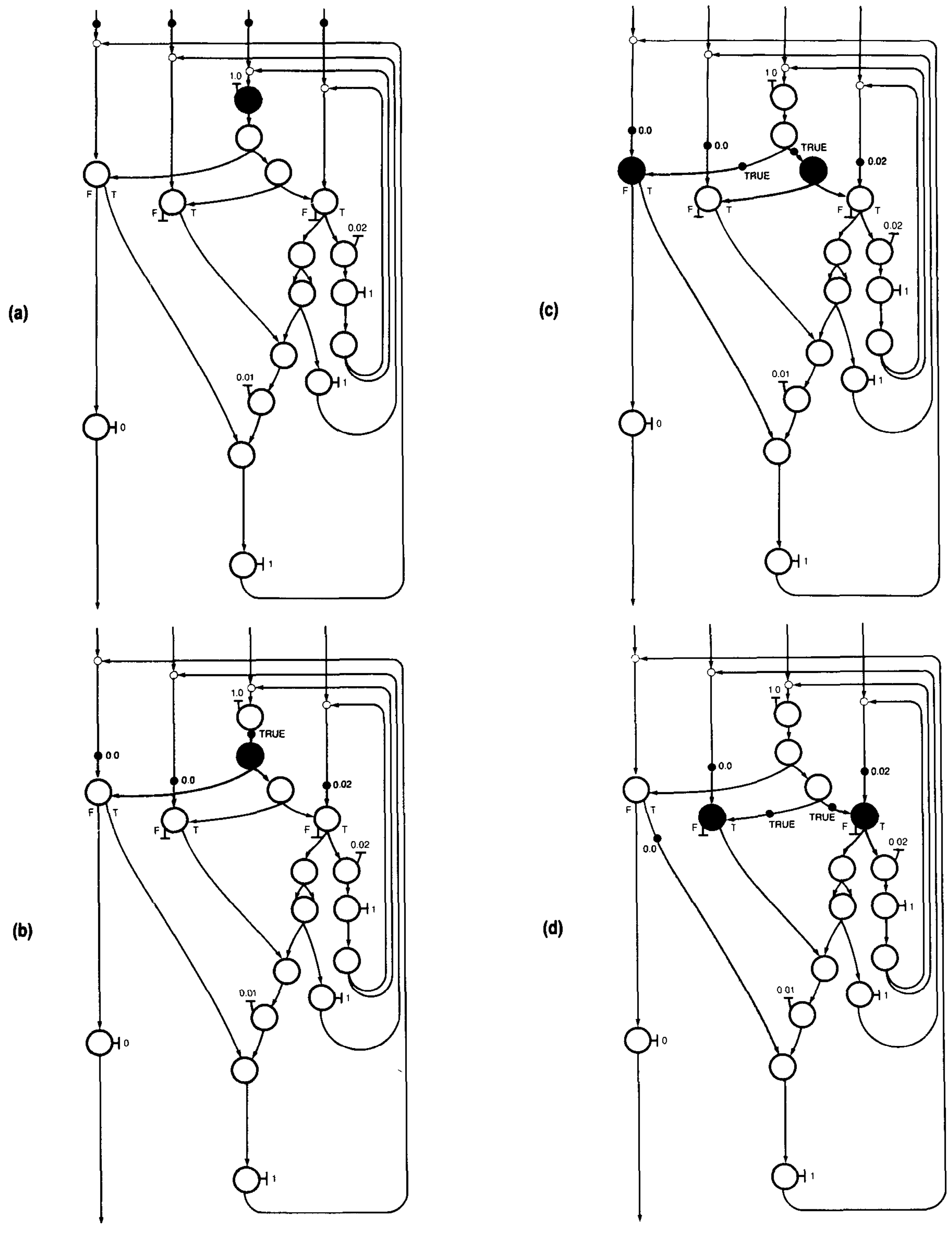
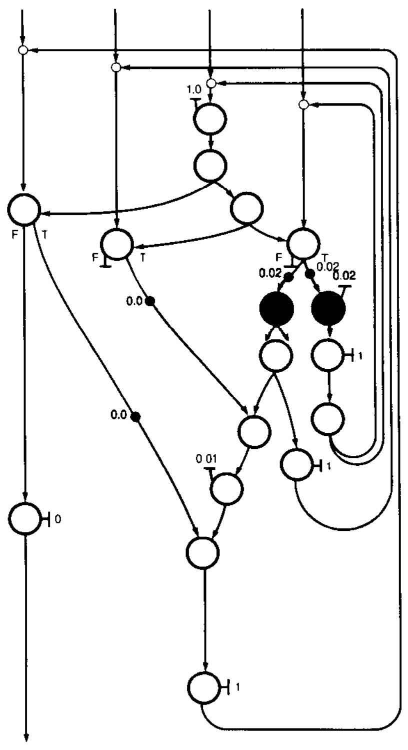


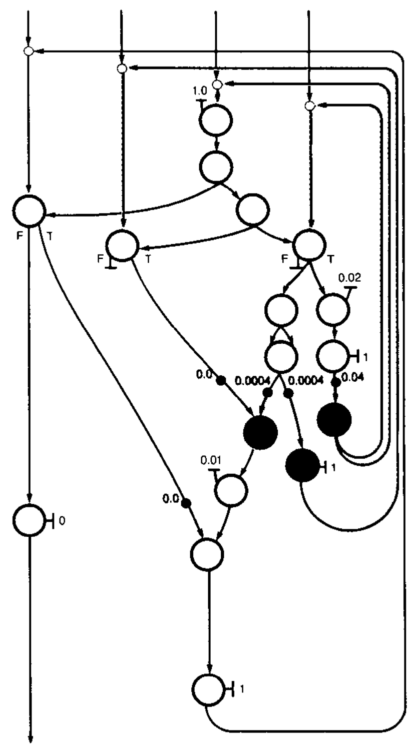
Figure 4 illustrates the way data appear to flow through the program graph during execution of the machine code. At the start of execution, the input data are presented in the form of data packets, known as tokens, on the input arcs of the graph. Execution then proceeds by transferring each token to the head of the arc on which it lies and executing any instruction that thereby receives a full complement of input tokens. The active arcs in each frame are shown in red, whereas the enabled instructions (i.e., those with a full complement of input tokens) and their output arcs (which will become active in the next frame) are shown in green. The transfer and execute cycle continues as shown until the output data have been sent and there is no further activity in the graph. Each token and instruction is considered in isolation

FIGURE 4. One Possible Execution Sequence for the Dataflow Problem in Figure 1

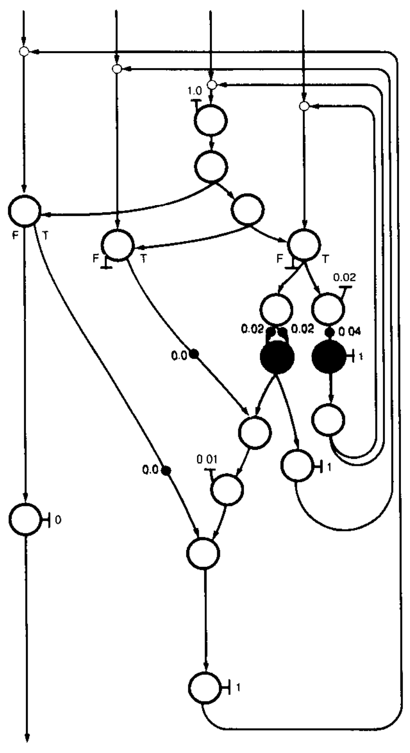
(e)



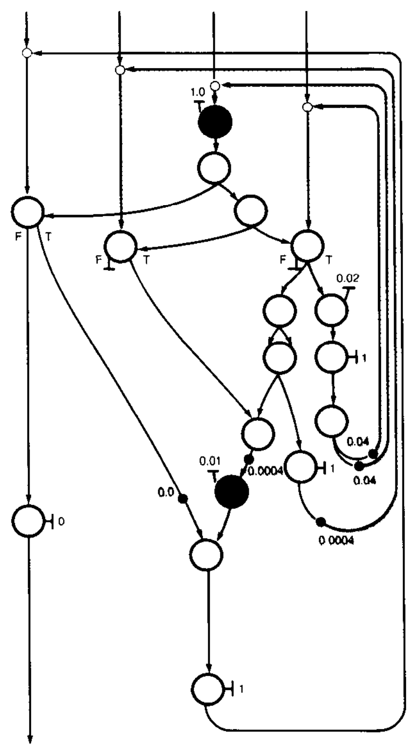
(g)



(f)



(h)



so that program execution is completely asynchronous. The required synchronization between communicating instructions is achieved by delaying execution of each instruction until all its input data are available. The process of determining that the input is ready is known as token-matching. At the end of each cycle of the program loop, the ADL instructions increment the iteration-level tag field so that tokens belonging to different cycles may be distinguished. A useful way of visualizing the effect of this operation is to imagine that each value of iteration level "colors" the tokens uniquely, so that only like-colored tokens can match with one another. This is illustrated by the tokens turning from black to blue as they pass from the first to the second iteration.

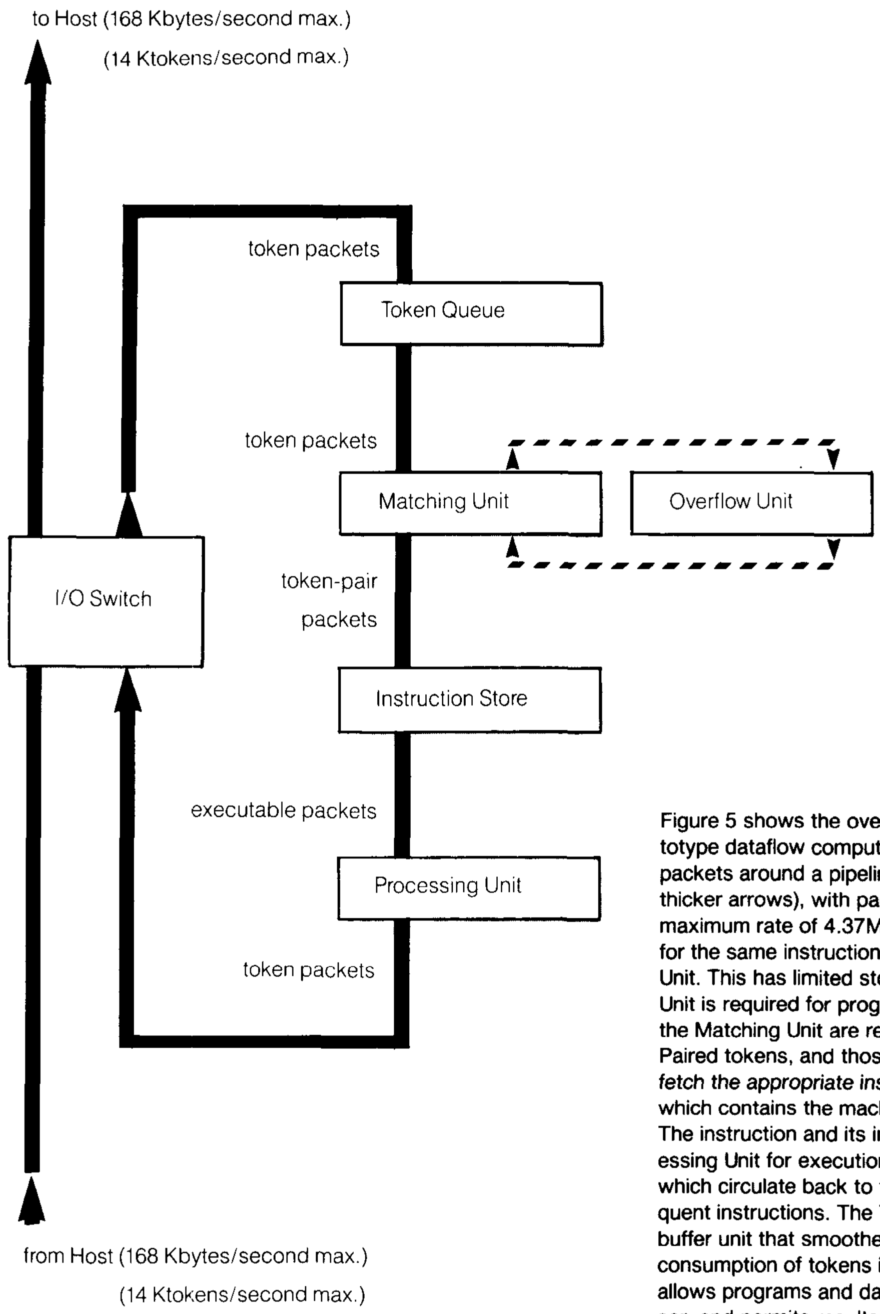


Figure 5 shows the overall structure of the Manchester prototype dataflow computer system. Tokens are carried in data packets around a pipelined ring structure (represented by the thicker arrows), with packets transferred between units at a maximum rate of 4.37M packets/second. Tokens destined for the same instruction are paired together in the Matching Unit. This has limited storage capacity, so that an Overflow Unit is required for programs with large data sets (the links to the Matching Unit are represented by the dashed arrows). Paired tokens, and those destined for one-input instructions, *fetch the appropriate instruction from the Instruction Store*, which contains the machine code for the dataflow program. The instruction and its input data are forwarded to the Processing Unit for execution. This produces further tokens, which circulate back to the Matching Unit to enable subsequent instructions. The Token Queue is a first-in-first-out buffer unit that smoothes out uneven rates of generation and consumption of tokens in the ring. The I/O Switch module allows programs and data to be loaded from a host processor, and permits results to be output for external inspection.

FIGURE 5. Manchester Dataflow System Structure

to iteration level) and SIL (set iteration level) in Figure 1 to ensure correct queuing of the loop termination control tokens at the inputs to the BRR (branch) instructions. Note also the use of explicit DUP (duplicate) instructions to replicate data required at two or more subsequent instructions. In order to limit the size of instructions, the Manchester system imposes a maximum fan-out from each instruction of two. Chains of duplicates can be used for larger fan-out. In some circumstances it is possible for a subsequent duplicate to be incorporated into the preceeding instruction (as in Figure 1 for the top-most MLR instruction). The maximum possible number of inputs to an instruction is also two; this has to do with the way tokens traveling to the same instance of an instruction are matched together. Manchester instructions are thus monadic or dyadic only. Certain monadic instructions are formed by dyadic operators with one fixed (literal) input (as also shown in Figure 1).

The BRR (branch) instructions act as "switches" in the arcs of program graphs and are used to implement conditionals, loops, and recursion. Each branch is controlled by a Boolean control input, which is shown entering the instruction from the side (usually, but not necessarily, the right-hand side). If the value of the token on this input is false, then the other incoming token (on the top input) is sent down the left-hand output (labeled F in Figure 1); otherwise the control is true, and the other input token is sent down the right-hand output (labeled T). Note that branch instructions can be used as "gates" that pass a value or destroy it, according to the Boolean control value, by leaving one of the output arcs unused (as also shown).

The process of executing a machine-level program is started by placing tokens representing the initial data values onto the input arcs of the program graph. Execution then proceeds by repeated application of the following graph execution rules:

1. Tokens travel (at any finite speed) toward the head of the arc on which they lie,
2. any instruction that has tokens ready at the head of all of its input arcs becomes enabled (i.e., ready to execute),
3. any enabled instruction may start execution as soon as there is a free instruction processor, and
4. executing instructions place output on their output arc(s) before terminating and releasing their processor for further executions (outputs from separate processors may be interleaved in any order).

Figure 4, on pages 38–39, illustrates the first steps of one possible execution sequence, based on these rules, for the dataflow program in Figure 1. In this sequence it is assumed that a large number of instruction processors are available so that all possible enabled instructions are executed simultaneously. It is also assumed that each instruction executes in one time step, regardless of the operation being performed. Different assumptions would produce alternative sequences of execution, but the same end results would always be produced. The way data seem to flow through the program graph during execution gives rise to the term "dataflow."

THE MANCHESTER DATAFLOW PROCESSOR

A block diagram of the prototype Manchester dataflow system is shown in Figure 5 on the preceding page. Figure 6a, on page 45, is a photograph of the system. The basic structure is a ring of four modules connected to a host system via an I/O Switch module. The modules operate independently in a pipelined fashion. Tokens are encapsulated in data packets that circulate around the ring. Token packets destined for the same instruction are paired together in the Matching Unit. This unit is organized as a two-tiered hierarchy with a separate Overflow Unit to handle large data sets. Paired tokens, and those destined for one-input instructions, fetch the appropriate instruction from the Instruction Store, which contains the machine-code for the executing program. The instruction is forwarded together with its input data to the Processing Unit, where it is executed. Output tokens are eventually produced and transmitted back toward the Matching Unit to enable subsequent instructions. The return path passes through the I/O Switch module, which connects the system to a host processor, and to the Token Queue, which is a first-in–first-out buffer for smoothing out uneven rates of generation and consumption of tokens.

The ring modules are independently clocked and are interconnected via asynchronous links capable of transferring large data packets at rates up to 10 million packets per second (represented by the thick arrows in Figure 5). This bandwidth is considerably higher than any that has been required by the modules yet constructed. The links to the Host system and the Overflow Unit are slower by a factor of about 500, although they are to be upgraded in the near future. The I/O Switch module is organized as a simple 2 × 2 common bus switch, which gives priority to input from the ring and selects the output route by performing a decode of certain marker bits. It has an internal clock period of 50 ns and is capable of transferring up to 5 million tokens/second. This rate is higher than the normal processing rates achieved by the other modules in the ring.

Figure 7, on page 46, illustrates the Token Queue and Matching Unit modules in detail. Figure 6b, on page 45, is a photograph of the Matching Unit module. The Token Queue comprises three pipeline buffer registers and a circular buffer memory. The token packets contained in the registers and store are 96 bits wide. The circular memory has a capacity of 32K tokens with 120 ns access time. The clock period is 37.5 ns, giving a maximum throughput of 2.67 million tokens/second. This is roughly equivalent to the processing rates achieved by the remaining ring modules. The discrepancies between the different module rates are due to the different engineering techniques used.

The Matching Unit contains six pipeline registers, a parallel hash table, and a 16-bit interface to the Overflow Unit. Each hash table board comprises a 64 Ktoken memory plus a 54-bit tag/destination comparator and interface control. There are 16 such boards at present, providing a 1Mtoken capacity, with space for expansion up to 1.25M tokens. Incoming tokens have a 16-bit hash function computed on their tag and destination fields as they are passed to the hash buffer register. The computed value is subsequently used to address the parallel hash table memory banks. Each bank compares its tag and destination contents with those of the incoming token, and a match causes the data field of the matching hash location to be output to the store buffer register along with the incoming token. The resultant token-pair packet is 133 bits wide, as shown in Figure 7. If there is no match between a stored token and the incoming token, the incoming token is written into the first free location accessed by that hash address. Overflows occur when all the accessed locations are occupied, in which case the nonmatching incoming token is sent to the Overflow Unit and indicator flags are set to notify subsequent tokens of this. Tokens that are destined for one-input instructions (such as "DUP" and

TABLE I. Maximum Average Match Rates versus the Proportion of Bypass Matching Operations

Pby	0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
Match rate (million matches/ second)	1.11	1.21	1.32	1.46	1.63	1.85	2.14	2.53	3.09	3.97	5.56

"SIL literal 0" in Figure 1) do not need to find partners and therefore bypass the hash memory access. Although bypass tokens do not search for a partner, each is counted as performing a "match" action in determining the processing rate of the Matching Unit.

The Matching Unit clock period is 180 ns, with a memory cycle time of 160 ns, giving "match" rates of 1.11 million matches/second for dyadic operators and 5.56 million bypasses/second for monadic operators. The average match rate thus depends on the proportion of executed instructions that receive only one input token. This proportion is known as the Pby (the proportion of bypass matching operations—see also the Program Characteristics section). Table I lists the maximum average match rates against the Pby (note that, in practice, the Pby is in the range 0.55 to 0.70).

The Overflow Unit is currently emulated by software in a microcomputer attached to the overflow interface. A special-purpose microcoded processor is under construction following the design shown in Figure 7. It will have an initial capacity of 32 Ktokens and will use linked lists accessed by a hash lookup. The target microcycle period is 250 ns, for a processing rate of up to 1 million matches/second.

Figure 8, on page 47, shows the detailed structure of the Instruction Store and Processing Unit modules. Figure 6c, on page 45, is a photograph of a typical board. The Instruction Store comprises two pipeline buffer registers, a segment lookup table, and a random-access instruction store to hold the program. The segment field of the incoming token-pair is used to access a segment descriptor from the segment table. This descriptor contains a base address for the segment and a maximum limit for offsets within the segment. The offset field of the incoming token is added to the base address and, provided the limit is not violated, the resulting address is used to access the instruction from the store. The instruction contents are 70 bits wide, as shown in Figure 8, and are substituted for the destination field of the input token-pair to form a 166-bit executable instruction package. This package is then forwarded for processing. The clock period for the Instruction Store is 40 ns, with a store access time of 150 ns, giving a maximum processing rate of 2 million instruction fetches per second.

The Processing Unit comprises five pipeline buffer registers, a special-purpose preprocessor, and a parallel array of up to 20 homogeneous microcoded function units with local buffer registers and common buses for input and output. The preprocessor executes those few global operations that cannot be distributed among the function units. These occur infrequently compared with the general opcodes, which pass straight through the preprocessor to be distributed to the first available function unit via the distribution bus. Each function unit contains a microcoded bit-slice processor with input and output buffering, 51 internal registers, and 4K words of writable microcode memory. The internal word length is 24 bits, with facilities for microcoding 32-bit floating-point arithmetic. Microinstructions are 48 bits wide. The function units compete to transmit their output onto the arbitration bus and thence out of the module. The Processing Unit clock has a period of 57 ns. The function unit microcycle period is 229 ns. The minimum time required to transmit 96 bits through a function unit is 13 microcycles, and the shortest instruction execution time (for DUP with one output) is 16 microcycles. This leads to a maximum instruction execution rate of 0.27 MIPS per function unit. To date, 14 function units have been used successfully to achieve processing rates of up to 2 MIPS (see the Evaluation Results section). With this complement of function units, the total software parallelism required to keep all the hardware busy is about 35-fold.

It will be noted that the host and overflow systems are much slower than the dataflow ring. This has had two ramifications: Either overflow of the matching store capacity or interaction with the host processor leads to a substantial drop in performance.

At present, the sole measurement that can be made of the system is of the interval between program start and the arrival at the host of the first output token. Programs are loaded in advance of their initial data. The data are then queued in the Token Queue, where reads are disabled until the last input token has been transmitted from the host. At this point Token Queue reads are enabled, and timing commences in the host: It will be halted by the first arrival from the output port of the Switch. Benchmark programs are usually organized to produce a single output token right at the end of their execution. By repeatedly running each program with different numbers of active function units, the speedup efficiency of the system can be assessed, as illustrated in The Benchmark Process section.

SYSTEM EVALUATION STRATEGY

There are three objectives for evaluation of the prototype hardware:

1. to tune the prototype hardware for optimum performance,
2. to determine the nature of software parallelism that

can be effectively exploited by the hardware, and
3. to determine the relative value of dataflow MIPS (compared to conventional MIPS).

The nature of the prototype hardware indicates that a three-phase approach to evaluation might be appropriate. The first phase is to assess performance for those programs that are small enough to execute entirely within the matching store limit (i.e., which do not generate overflow requests). This is the phase reported below. It comprises three subphases:

1. plotting speedup curves,
2. interpreting the results, and
3. rectifying any discovered hardware problems.

The second evaluation phase will involve analysis of programs that generate moderate amounts of overflow. Bottlenecks in the overflow loop will eventually be identified and subsequently rectified, although this cannot be undertaken with the existing overflow processor system. The third phase involves the development of a hierarchical memory to cope with programs that generate enormous quantities of overflow. This is regarded as a longer term objective, which will be addressed initially through the Structure Store Unit discussed in the Future Directions section.

For the evaluation that follows, analysis is restricted to overflow-free programs, although many other characteristics of the codes have been varied. These characteristics were measured by means of a crude software simulator for the dataflow system.

PROGRAM CHARACTERISTICS

In order to measure program characteristics, a dataflow simulator that makes many simplifying assumptions about the system architecture is used. The principal assumptions made are

1. that each instruction executes in the same time (execution therefore proceeds in discrete equal time steps),
2. that an unlimited number of function units can be used during any one time step, and
3. that output from any executed instruction can be transmitted to an enabled successor instruction within the execution time period.

Of course these are somewhat unrealistic assumptions, but they are helpful in making an approximate characterization of each program.

The two fundamental time measurements recorded for each program are S1, the total number of instructions executed (which would be the number of time steps required if only one function unit was available), and Sinf, the number of simulated time steps required (with an unlimited number of function units permanently available). The ratio S1/Sinf = avePara gives a crude measure of the average parallelism available in the program. A more comprehensive trace of the time variance of program parallelism can be obtained if needed.

The simulator also records utilization of the system memories, as follows:

Codesize	=	the size of the machine-code program (in 9-byte instructions),
maxTQsize	=	the maximum occupancy of the Token Queue circular buffer store (in 12-byte tokens), and
maxMSsize	=	the maximum occupancy of the Matching Store hash table (also in 12-byte tokens).

The proportion—the Pby—of executed instructions that bypass the matching store is also recorded. This corresponds to the fraction of one-input instructions executed. An important measure of performance for numerical computation is the execution rate expressed in MFLOPS (million floating-point operations per second). Different machine architectures and programming systems can be compared by measuring their respective MIPS to MFLOPS ratios. Consequently, this ratio is recorded by the simulator.

Looking at one cycle of the integration program in Figure 1, it can be easily seen that S1 = 16. It is not immediately obvious that Sinf = 7, but this can be checked by locating the longest cycle of dependent instructions (i.e., that forming the value of x, which is input to the CGR instruction). Simulation of 50 cycles (i.e., the complete program of Figure 3) gives S1 = 808 and Sinf = 356. Consequently, the average parallelism, avePara, is 2.3. The total Codesize is 17 instructions (153 bytes), maxTQsize is 5 tokens (60 bytes), and maxMSsize is 3 tokens (36 bytes). The Pby is 0.625, and the ratio MIPS/MFLOPS is 2.7 (i.e., 2.7 instructions are executed on average for every useful floating-point operation).

For comparison, the code compiled from the SISAL version of the integration program, shown in Figure 2, produces the following characteristics: S1 = 2455, Sinf = 829, avePara = 3.0, Codesize = 80 (720 bytes), maxTQsize = 11 (132 bytes), maxMSsize = 15 (180 bytes), Pby = 0.628, and the MIPS/MFLOPS ratio = 8.1. This comparison gives a rough indication of the relative efficiencies of compiled and hand-written code. For both programs, DUP (duplicate) accounts for 25 percent of all executed instructions.

THE BENCHMARK PROCESS

A total of 14 benchmark programs with 29 different input data sets has been analyzed for the following performance evaluation. The programs are listed in Table II, along with their characteristics, as measured by a simulator. A variety of problem types is represented, and several source languages have been used. MAD is a single-assignment language like SISAL, and MACRO is an intermediate-level language like TASS. The effect of program parallelism has been assessed for both similar and distinct programs. Parallelism for each particular code was varied by adjustment of the input data values. Many different patterns of time variance of parallelism

TABLE II. Approximate Characterization of the Operational Behavior of Programs

[illegible] have table.

Name	Source	Codesize (* 9 bytes)	tot. inst. S1 executed	# simulated Sinf time steps with ∞ processors	average parallelism avePara	proportion bypass Pby matching store	token queue maxTQsize (* 12 bytes	matching store maxMSsize ——)	MIPS/ FU	MIPS/ MFLOPS
LAPLACEA/1	MACRO	58	290,112	2,167	134	0.70	449	640	0.140	15.9
LAPLACEA/2	MACRO	58	567,200	2,707	210	0.70	701	1,000	0.140	15.6
SUM/1	MAD	107	30	17	2	0.67	6	4	0.165	—
SUM/2	MAD	107	402	79	5	0.62	22	41	0.170	—
SUM/3	MAD	107	1,208	120	10	0.61	56	99	0.171	—
SUM/4	MAD	107	2,820	141	20	0.61	109	220	0.172	—
SUM/5	MAD	107	9,082	182	50	0.61	418	766	0.172	—
SUM/6	MAD	107	20,428	204	100	0.61	824	1,515	0.172	—
SUM/7	MAD	107	44,980	225	200	0.61	1,626	3,293	0.172	—
SUM/8	MAD	107	123,472	247	500	0.61	6,058	11,279	0.172	—
INTEGRATE	MAD	263	2,051	166	12	0.64	41	106	0.158	39.4
FFT/1	MACRO	606	13,989	264	53	0.70	211	794	0.112	11.7
FFT/2	MACRO	606	14,086	264	53	0.70	211	794	0.112	11.7
FFT/3	MACRO	606	15,374	569	27	0.69	152	1,168	0.111	11.1
FFT/4	MACRO	606	32,661	310	105	0.70	419	1,834	0.109	11.3
MATMULT	SISAL	657	100,288	425	236	0.58	6,001	15,074	0.100	50.1
LAPLACEB	SISAL	811	191,984	915	210	0.60	1,987	15,744	0.114	37.0
PLUMBLINE1	MACRO	828	7,531	156	48	0.61	208	262	0.173	—
PLUMBLINE2	MAD	1,462	19,076	908	21	0.56	299	484	0.131	—
GAUSS	SISAL	3,201	215,723	3,620	60	0.57	1,260	18,457	0.106	—
LOGICSIM/1	MACRO	3,819	64,660	1,227	53	0.63	339	905	0.175	—
LOGICSIM/2	MACRO	3,819	348,700	5,067	69	0.63	1,779	3,785	0.175	—
SPLICE	SISAL	6,957	5,031,909	165,647	30	0.69	658	6,921	0.111	—
RSIM/1	SISAL	23,996	189,746	12,611	15	0.61	147	1,894	0.140	—
RSIM/2	SISAL	24,314	1,135,912	62,563	18	0.60	259	3,563	0.137	—
RSIM/3	SISAL	24,477	851,137	50,866	17	0.60	403	2,922	0.139	—
RSIM/4	SISAL	24,850	1,108,104	54,048	20	0.62	691	4,437	0.128	—
SIMPLE	SISAL	26,385	519,501	8,194	63	0.59	1,254	9,635	0.117	39.0
IV/1	SISAL	39,091	126,991	6,571	19	0.62	561	3,711	0.117	—

These data were obtained from software simulation of the Manchester prototype dataflow machine. The simulator imitates sequential execution of programs but also keeps track of the shortest path through the graph, making the assumption that each instruction could be executed in an identical time period. The ratio of the total number of instructions executed (S1) to the length of the shortest path (Sinf) gives a rough measure of the amount of parallelism in the graph (avePara). Store usage is recorded as the maximum simulated store requirement for the Token Queue (maxTQsize) and the Matching Store (maxMSsize), assuming that neither store overflows. The final recorded characteristic is the proportion (Pby) of one-input instructions executed, since these bypass the Matching Unit, which constitutes the major bottleneck in the ring. The variation of parallelism with time is not accounted for since it appears to be unimportant in predicting the speedup obtained when additional parallel resources are used to execute the program. The characteristics of a variety of benchmark programs that have been executed on the prototype Manchester dataflow system are listed. The smaller programs have been written in a macroassembler language, MACRO, which was a forerunner to the template assembler, TASS. The larger programs have been written in the high-level single-assignment languages MAD and SISAL. The final columns record hardware execution characteristics for the benchmark programs. The first of these (MIPS/FU) shows the average processing rate with a single active function unit. This value is related to the average number of microinstructions executed per machine instruction. Low values imply the use of many complex operators, such as the floating-point trigonometric functions. The final column (MIPS/MFLOPS) is recorded for programs that make heavy use of floating-point arithmetic and indicates the average number of instructions executed per "useful" floating-point operation.

were found. In addition, code sizes and store occupancies varied considerably. Note, however, that the Pby is in the range 0.56 to 0.70 for all programs.

As mentioned above, the only measurement that can be made on the prototype hardware is the execution time until the arrival at the host of the first output token with a given number of active function units. For n function units this time is denoted Tn, where n has been varied from 1 to 14. Knowing the simulator-derived characteristics of each program, the following quantities can be derived from the values of Tn:

$Pn = T1/Tn$:
: the effective number of function units when n is active,

$En = 100Pn/n$:
: the percent utilization of n active function units,

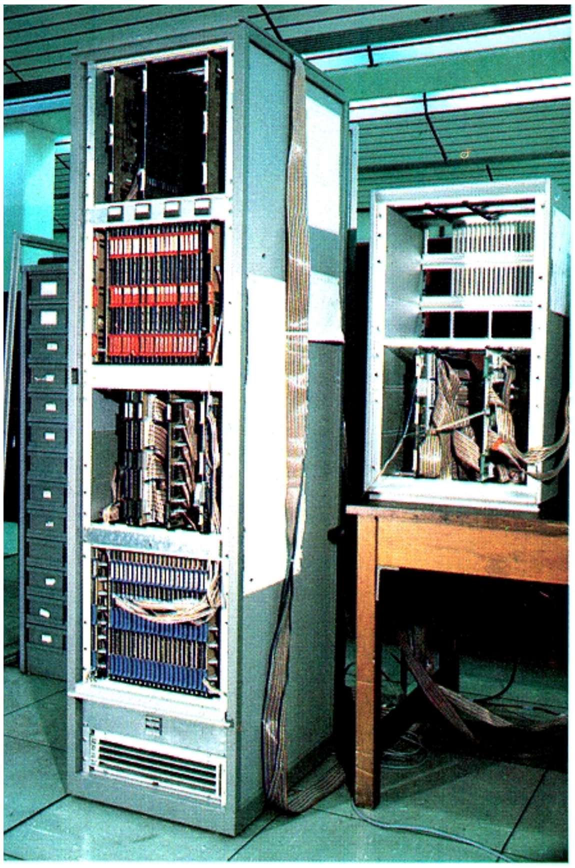


FIGURE 6a. The Manchester Prototype Dataflow Computer

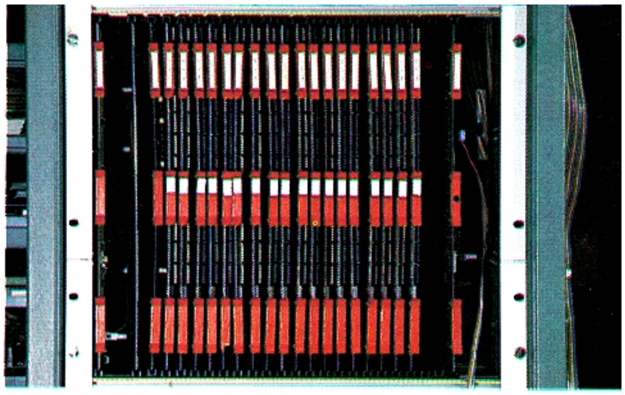


FIGURE 6b. The Matching Unit Module

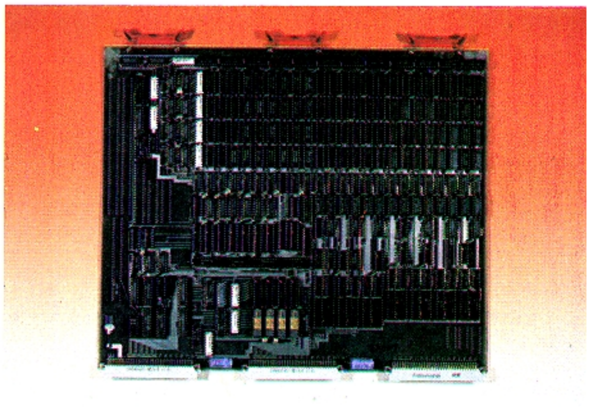


FIGURE 6c. A Typical Board

$Mn = S1/Tn$:
the actual MIP rate of n active function units, and
$Mn' = nS1/T1$:
the potential MIP rate of n active function units.

A typical set of measurements (for the SIMPLE program) is shown in Table III.

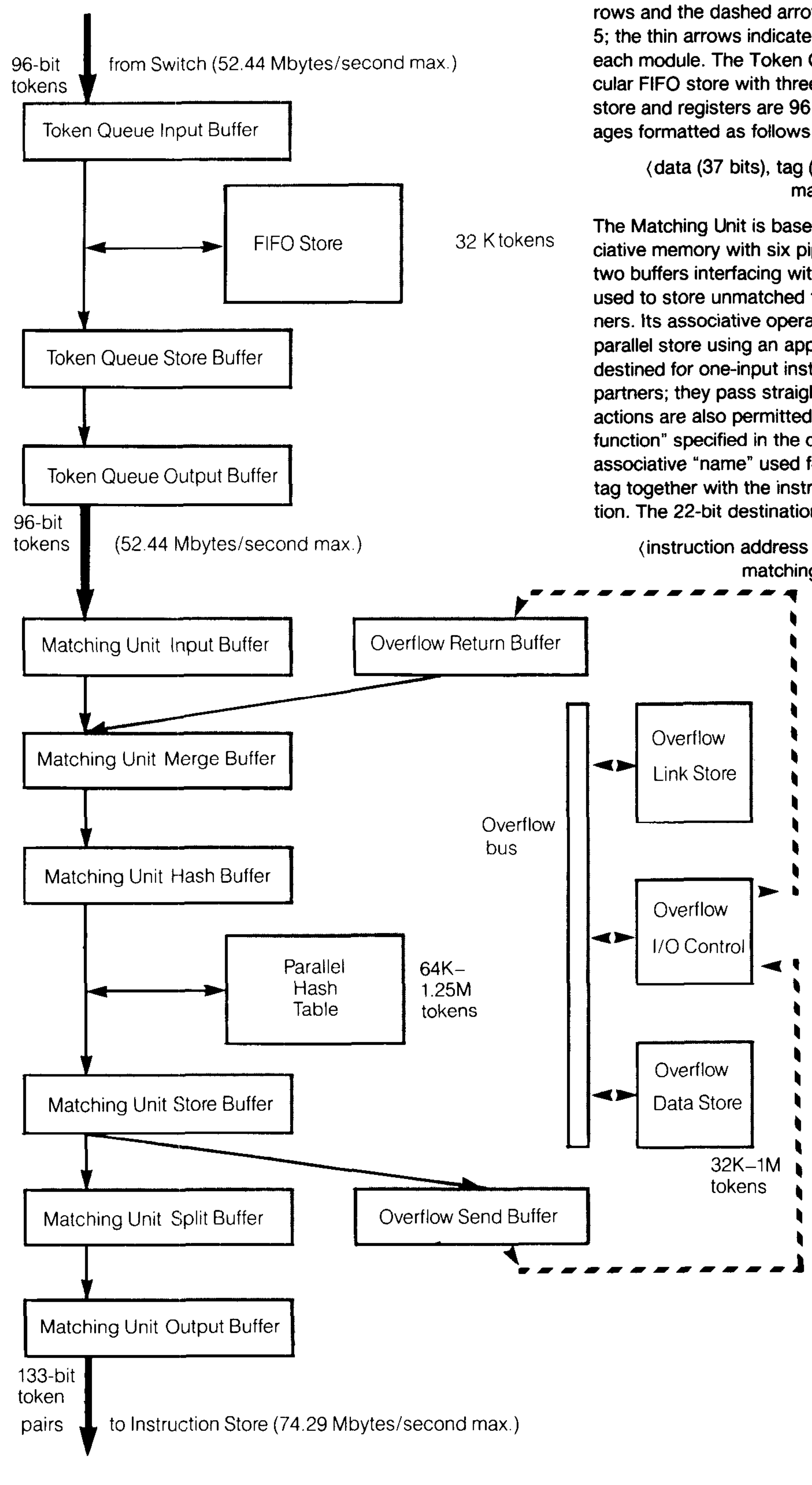
EVALUATION RESULTS

For each program and data set run together, the values of Tn, Pn, En, Mn, and Mn' were tabulated. To interpret these values as measures of the speedup performance of the system, Pn is plotted against n, as shown in Figure 11 (for the RSIM/1 program). Notice how lines of constant function unit utilization appear on this graph. To compare the results for different kinds of programs, the results are better presented after normalization by a factor $S1/T1 = M1$. This entails plotting Mn against Mn' (actual MIPS versus potential MIPS), as also shown in Figure 9, on page 48.

The shape of the speedup curve is typical of the results obtained when parallelism in a program is limited. There is an initial portion in which speedup is nearly

TABLE III. A Typical Set of Measurements of the Execution Time that Elapses before the First Output Token Arrives at the Host

Function units (n)	Run time (seconds) (Tn)	Speedup (Pn)	Efficiency (En)	Actual MIPS (Mn)	Potential MIPS (Mn')
1	4.4215	1.00	100.0	0.117	0.117
2	2.2106	2.00	100.0	0.235	0.235
3	1.4751	3.00	99.9	0.352	0.352
4	1.1077	3.99	99.8	0.469	0.470
5	0.8886	4.98	99.5	0.585	0.587
6	0.7429	5.95	99.2	0.699	0.705
7	0.6400	6.91	98.7	0.812	0.822
8	0.5643	7.84	97.9	0.921	0.940
9	0.5071	8.72	96.9	1.024	1.057
10	0.4629	9.55	95.5	1.122	1.175
11	0.4301	10.28	93.5	1.208	1.292
12	0.4038	10.95	91.3	1.287	1.410


96-bit tokens
from Switch (52.44 Mbytes/second max.)
Token Queue Input Buffer
FIFO Store
32 K tokens
Token Queue Store Buffer
Token Queue Output Buffer
96-bit tokens
(52.44 Mbytes/second max.)
Matching Unit Input Buffer
Overflow Return Buffer
Matching Unit Merge Buffer
Overflow Link Store
Overflow bus
Matching Unit Hash Buffer
Overflow I/O Control
Parallel Hash Table
64K–1.25M tokens
Matching Unit Store Buffer
Overflow Data Store
32K–1M tokens
Matching Unit Split Buffer
Overflow Send Buffer
Matching Unit Output Buffer
133-bit token pairs
to Instruction Store (74.29 Mbytes/second max.)

The internal structure of some of the hardware modules in the Manchester prototype machine are shown. The thick arrows and the dashed arrows correspond to those in Figure 5; the thin arrows indicate data paths that are internal to each module. The Token Queue comprises a 32K-word circular FIFO store with three surrounding buffer registers. The store and registers are 96 bits wide and contain token packages formatted as follows:

⟨data (37 bits), tag (36 bits), destination (22 bits), marker (1 bit)⟩

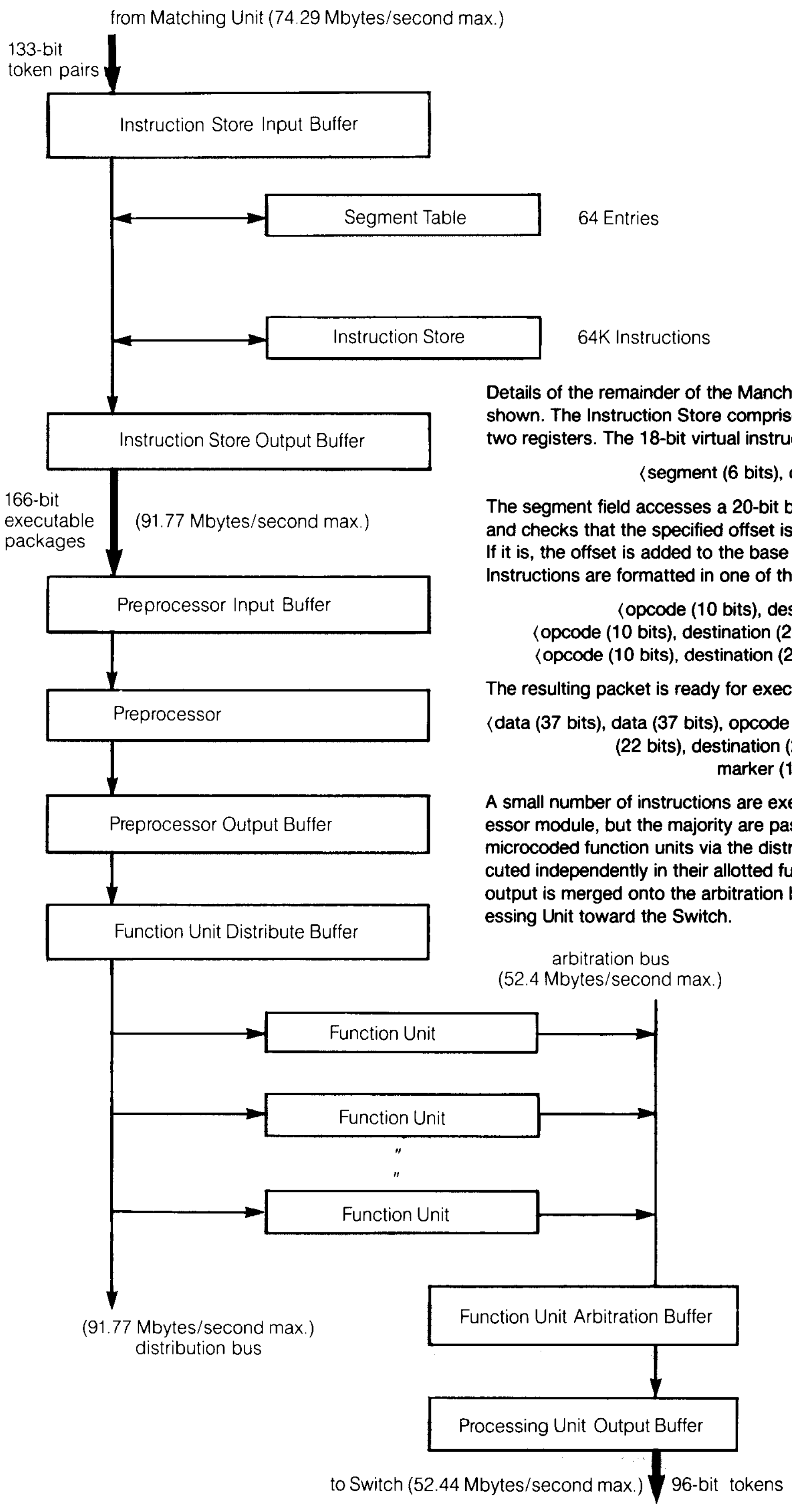
The Matching Unit is based on a 1.25M-word pseudoassociative memory with six pipeline registers in the main ring and two buffers interfacing with the Overflow Unit. The memory is used to store unmatched tokens while awaiting their partners. Its associative operation is achieved by accessing a parallel store using an appropriate hash function. Tokens destined for one-input instructions do not need to match with partners; they pass straight through the unit. Other matching actions are also permitted, according to the 3-bit "matching function" specified in the destination field of the token. The associative "name" used for matching comprises the token's tag together with the instruction address part of the destination. The 22-bit destination field is therefore split as follows:

⟨instruction address (18 bits), left/right input (1 bit), matching function (3 bits)⟩

The Overflow Unit handles tokens that cannot be placed in the parallel hash table because they encounter a full hash entry. There is no attempt to compute a new hash function since it does not matter if subsequent tokens match with their partners before the overflowing token matches with its partner. The asynchronous nature of the dataflow model ensures that the computation will yield determinate results regardless of the order in which tokens are matched. Overflow tokens are stored in linked lists in the Overflow Unit, which contains a microcoded processor together with data and pointer memories. Matched token pairs are sent out in packets with the following format:

⟨data (37 bits), data (37 bits), tag (36 bits), destination (22 bits), marker (1 bit)⟩

FIGURE 7. A Close-Up Look at the Token Queue, Matching Unit, and Overflow Unit


from Matching Unit (74.29 Mbytes/second max.)
133-bit token pairs
Instruction Store Input Buffer
Segment Table
64 Entries
Instruction Store
64K Instructions
Instruction Store Output Buffer
166-bit executable packages
(91.77 Mbytes/second max.)
Preprocessor Input Buffer
Preprocessor
Preprocessor Output Buffer
Function Unit Distribute Buffer
arbitration bus
(52.4 Mbytes/second max.)
Function Unit
Function Unit
"
"
Function Unit
(91.77 Mbytes/second max.)
distribution bus
Function Unit Arbitration Buffer
Processing Unit Output Buffer
to Switch (52.44 Mbytes/second max.)
96-bit tokens

Details of the remainder of the Manchester prototype machine are shown. The Instruction Store comprises a random-access memory and two registers. The 18-bit virtual instruction addresses have the format

⟨segment (6 bits), offset (12 bits)⟩

The segment field accesses a 20-bit base address in the segment table and checks that the specified offset is within the limits of the segment. If it is, the offset is added to the base and used to read the instruction. Instructions are formatted in one of the following forms:

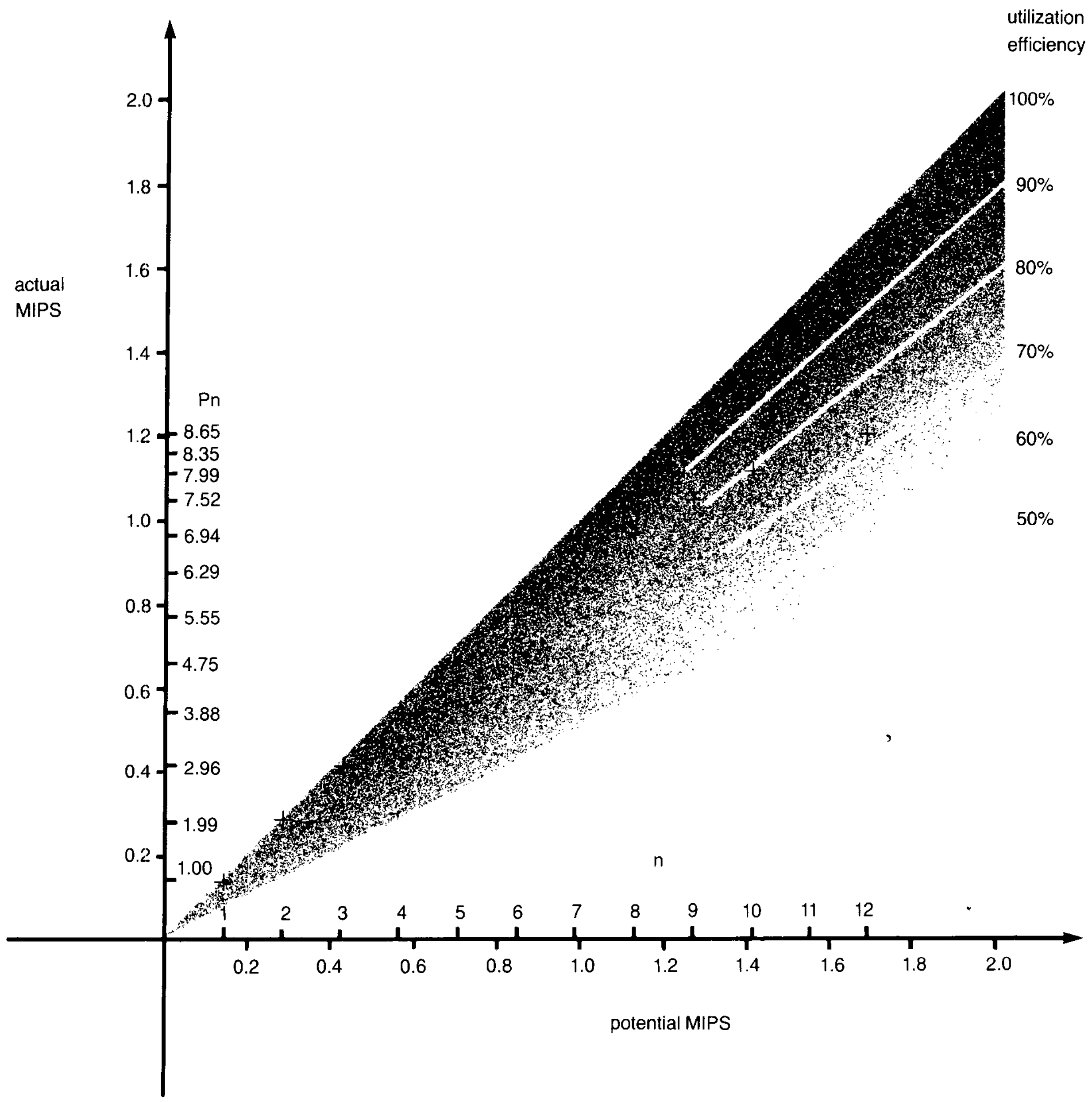
⟨opcode (10 bits), destination (22 bits)⟩
⟨opcode (10 bits), destination (22 bits), destination (22 bits)⟩
⟨opcode (10 bits), destination (22 bits), literal data (37 bits)⟩

The resulting packet is ready for execution and is in the form

⟨data (37 bits), data (37 bits), opcode (10 bits), tag (36 bits), destination (22 bits), destination (22 bits (optional)), marker (1 bit)⟩

A small number of instructions are executed in the specialized preprocessor module, but the majority are passed into one of the homogenous microcoded function units via the distribution bus. Instructions are executed independently in their allotted function unit, and the eventual output is merged onto the arbitration bus and thence out of the Processing Unit toward the Switch.

FIGURE 8. Details of the Instruction Store and Processing Unit



The speedup obtained when additional hardware parallelism is introduced into the prototype ring by allowing extra function units to participate in executing the program RSIM/1 is shown. The curves are obtained by measuring the execution time (*Tn*) associated with the use of *n* function units. They can be interpreted in several different ways. First, they show the effective processing rate of *n* function units (*Pn*), normalized so that *P*1 = 1.0. Second, they demonstrate the efficiency of utilization of *n* function units. This rate should ideally be constant at 100 percent to provide linear speedup as extra function units are added, but in practice it decreases as *n* increases. Third, the curves give the absolute processing rate (in MIPS) achieved by a system with *n* function units for each program.

FIGURE 9. A Plot of the Speedup Performance of the Prototype

linear in n (and where En is thus close to 100 percent), followed by a gradual deterioration in utilization until a program-constrained limit is reached. In the case of the RSIM/1 program, avePara is only 15-fold, so the effective use of between 8 and 9 function units when 12 are active is acceptably efficient.

It is, of course, possible for a highly parallel program to reach a hardware-constrained limit before it runs out of program parallelism. The effect of this behavior, viewed on speedup curves such as in Figure 9, will be similar to the software-limited case described above, except that the limit will be imposed by the match rate achieved in the Matching Unit, as shown in Table I. With the maximum number of function units limited to 14, program runs did not reach the current maximum match rate of around 2 MIPS, and so this effect was not observed.

In another work,[1] we have published superimposed speedup curves for many of the benchmark programs listed in the previous section. These curves show the effects of variable instruction mix and variable program parallelism. It is noticeable that when floating-point instructions (which are microcoded in the function units and hence take much longer to execute than integer operations) are used, the potential MIP rate for each function unit is correspondingly smaller. However, the major pattern to emerge from this study is the importance of the parameter avePara in determining the shape of the speedup curve for various programs. Programs with similar values of avePara exhibit virtually identical speedup curves. The higher the value, the closer the curve is to the 100 percent utilization rate. This seems to indicate that this crude approximation to the overall average parallelism of a code is all that is necessary for an accurate prediction of its speedup curve. This applies regardless of factors such as time variance of parallelism, the source language used, the proportion of one-input instructions executed, etc. It is surprising that such a simple measure should give such a constant indication of the pattern of use of processing resources, but it does help to answer the question of what nature a program has to have if it is to be suitable for execution on this dataflow system. A program is suitable if it has a value of avePara in the region of 40 or more. Significantly, the larger applications codes exhibit the same patterns as the simpler benchmarks.

Another noticeable feature of the study is that there is an unusable area of potentially high function unit utilization above an execution rate of about 1 MIPS. Since this occurs for programs with large values of avePara, it seems unlikely that this performance area has been lost because of a lack of program parallelism, and so other causes have been sought.

One suggestion is that the use of multiple-function units in a pipeline causes contention problems in the Processing Unit arbitrator and thus leads to performance degradation. Another possibility is that disparate execution times in the pipeline stages lead to pipeline "starvation," a well-known cause of performance degradation. Two experiments were designed to determine the actual cause.

The first experiment confirmed that programs are not responsible for restricting available concurrent activity. The method adopted was to take a well-understood program (the double-recursive SUM code) and force it into a highly parallel form by artificially excluding those parts known to be serial in nature. In this program the serial sections occur at the start and end of each run. They can be eliminated by subtracting the run times for two large, but different, data sets. The data sets chosen were those that generated individual avePara values of 80 and 150. The timing that results is for a simulated code that has an overall avePara value greater than 700, with no serial sections.

The second experiment was designed to eliminate the effects of pipeline starvation caused by unsatisfied match requests in the Matching Unit. This was achieved by running a test program with Pby = 1, in which all instructions have one input and the Matching Unit is always bypassed. In this mode the Matching Unit can process tokens at a rate equivalent to nearly 6 MIPS, and it can be guaranteed that the processing rate is limited solely by the number of available function units.

The results of these experiments show three things. First, the performance degradation above the 1-MIPS execution rate occurs even when the effect of serial code has been eliminated. Second, programs that always bypass the Matching Unit are able to enter the "forbidden" zone. Third, where parallelism is limited solely by software, the totally flat curves exonerate the Processing Unit arbitrator because they show that performance is never degraded when function units are added.

The implication of these results is that there must be a deficiency in the pipeline buffering between the Matching Unit and the Processing Unit. The system cannot cope with prolonged sequences of unsuccessful match operations without starving the function units of input. It has subsequently been established that additional buffering at the output of the Matching Unit significantly reduces the falloff in speedup curves for highly parallel programs.

It is not clear whether an average instruction executed in a dataflow system is more or less powerful than an average conventional instruction. This casts some doubt on the value of the MIPS rates quoted above. Consequently, the relative value of dataflow MIPS has been assessed by studying the MIPS/MFLOPS ratios obtained for various programs. These ratios have been measured for high-speed conventional systems, such as the CDC6600, CDC7600, and Cray-1, by users, such as Lawrence Livermore National Laboratory, who have large floating-point computational requirements. It has been discovered that assembly-language program-

[1] Gurd, J. R., and Watson, I. A preliminary evaluation of a prototype dataflow computer. In *Proceedings of the Ninth IFIPS World Computer Congress*, R.E.A. Mason, Ed. Elsevier North-Holland, New York, Sept. 1983.

mers for such systems can achieve between three and four MIPS/MFLOPS, whereas good FORTRAN compilers achieve between five and seven MIPS/MFLOPS. The corresponding ratios for the integration program of Figures 2 and 3 (2.7 for assembler and 8.1 for SISAL) indicate that the measured dataflow MIPS have the potential to match the power of conventional-sequential MIPS. However, ratios for larger SISAL programs are often much bigger than this, ranging from 20 to 50. This indicates that present compilation techniques require considerable improvement.

This opinion is reinforced by a comparison of the dataflow results with the run times achieved for conventional implementations of some of the benchmark programs described above. For example, Table IV compares the dataflow run times for the RSIM family of programs with those obtained for versions written in the C language and executed on a VAX11/780 system. It can be seen that the current SISAL/dataflow system is about five to ten times slower than the C/VAX11/780 system.

More of these direct comparisons are being made between the dataflow system and conventional machines. They involve two categories of competitive run-time measurement for a range of benchmark programs. The first category uses single-source programs, written in SISAL, to evaluate different SISAL implementations. The second allows rewriting of programs, to assess the impact of code optimization in different language systems. The most useful comparisons will be with similar-sized sequential systems, such as the VAX 11/780. The VAX SISAL compiler, expected to be ready in early 1985, will enable comprehensive single-source tests to proceed. Tests in the second category await the translation of more programs from conventional languages into SISAL.

FUTURE DIRECTIONS

For the immediate future, the results presented here should provide ample motivation for improving the efficiency of the generated code for the SISAL/dataflow system. This objective will be pursued with a combination of software and hardware enhancements to tackle inefficiencies in the compiler system and in the machine architecture. It is believed that system performance will exceed that of conventional language systems on the VAX11/780 for a variety of applications within the next year. In the longer term, it should be possible to use the extensible nature of the dataflow hardware to provide much higher computing rates by building a dataflow multiprocessor. We now consider these intended improvements and the benefits we expect them to provide. Implementation of all these various enhancements should significantly improve the SISAL/dataflow system performance reported earlier.

TABLE IV. Comparison of VAX and Dataflow Run Times for RSIM Programs (all run times in seconds)

Program	VAX11/780	Dataflow (1FU)	Dataflow (12FUs)
RSIM/1	0.04	1.36	0.16
RSIM/2	0.10	8.26	0.89
RSIM/3	0.08	6.12	0.68
RSIM/4	0.28	8.67	0.88

Improvements to the code generation system are being made by letting the SISAL compiler implementation influence the design of the dataflow instruction set. Frequently occurring combinations of instructions are being amalgamated into new "super" instructions, with attendant reduction in S1 and Sinf parameters and improved execution speed. For example, the introduction of the SAZ (set activation name and zero index) instruction reduced Codesize and S1 by about 10 percent for most programs.

Improvements can also be realized through more conventional optimization techniques, such as common subexpression elimination, removal of constants from loops, etc. Researchers at Lawrence Livermore have implemented several such optimizations for an intermediate phase of the SISAL compiler, and these also reduce Codesize and S1 by about 10 percent.

Experience with the larger benchmark programs indicates that the overhead associated with storing data structures in the Matching Unit is excessive. Each stored token carries its tag and destination individually, which leads to replication of information that should be compacted. Two schemes have been proposed to alleviate this waste. The first involves the creation of a matching store hierarchy, using a scheme analogous to a conventional paging system. This is difficult to design unless it proves consistently feasible to identify areas of locality in dataflow programs. With the present state of knowledge, this cannot be guaranteed. Consequently, an alternative scheme involving the construction of a specialized Structure Store Unit has been adopted. This unit will be attached to the processing ring by a second Switch module located between the Processing Unit and the I/O Switch. A prototype implementation should be operational early in 1985.

The effect of a Structure Store Unit on system performance has been studied using an enhanced version of the simulator described in the Program Characteristics section. The programs used were compiled from the SISAL language using a modified compiler. For a typical program, S1 is reduced by about 40 percent. Much of this improvement results from the removal of spurious parallelism, causing the overall parallelism to drop slightly.

Unfortunately, the amounts of Matching Unit and Token Queue store used are high, whether or not the Structure Store Unit is used. It is therefore important to assess matching store usage and to optimize the handling of Matching Unit overflows. Studies in this area are hampered by the slow speed of the current host system and overflow processor interfaces, and so upgraded versions of these are being installed. A longer-term project would involve investigating more general ways of reducing the amount of matching store and Token Queue store required for a computation. This would require the design of an "intelligent" Token Queue that could schedule sections of highly parallel programs in such a way as to minimize these storage requirements. Preliminary studies of recursive divide-and-conquer algorithms indicate that there are enormous potential savings in this area.

It is not feasible to add extra function units to the Processing Unit indefinitely, since the match rate in the Matching Unit will eventually limit the processing rate. An important objective of research into dataflow architecture is thus to establish techniques for constructing and utilizing multiprocessor systems in which the matching process is distributed. In a dataflow multiprocessor, a number of processing rings are connected together via an interconnection network. The network allows any ring to send results to any other ring. The choice of network is critical to large system performance. Some networks have an equal delay for all communications; other networks penalize some transfers more than others. There is also a relationship between the power of each processor and the total size of the network. Some systems emphasize simplicity of processor design and thus require large communications nets. Other systems have powerful processors and therefore need smaller networks to achieve the same overall computing power. The Manchester system falls into the latter category.

There are no present plans to construct a Manchester multiprocessor using the present system design and technology. However, such a system would be attractive if the basic processing ring could be implemented in a higher density VLSI technology. Consequently, simulation of a multiprocessor based on the existing processing ring has been undertaken. Performance results for systems containing up to 64 processing rings have shown respectable speedups for some of the benchmark programs reported above.

The key requirement for high performance in any multiprocessor structure is uniform distribution of work across the processors. This can be achieved at compile/link time, but often requires intervention from the applications programmer and always requires knowledge of the system configuration. It is more desirable to achieve the distribution automatically, whereby the compiler/linker would not need to know the configuration. The fine grain of parallelism in dataflow would facilitate this, although other substantial problems persist. Various load/run-time "split functions" have been investigated to distribute the work load. These use hashing techniques similar to those used in the pseudoassociative Matching Unit.

It should be noted that the ability to use this kind of "randomizing," postcompilation split function constitutes the major advantage of dataflow over more conventional, coarse-grain multiprocessors. In the latter system it is necessary for the programmer to direct the distribution of code across the processors, since load-time splitting is currently too inefficient and expensive. It may be that future research will uncover automatic coarse-grain split methods, but it is by no means clear that this can be achieved.

CONCLUSIONS

The Manchester project has constructed an operational tagged-token dataflow processor large enough to tackle realistic applications. A small range of benchmark programs has been written and executed on the hardware to provide evaluation data. The preliminary evaluation has returned several important results: First, it has established that a wide variety of programs contains sufficient parallelism to exhibit impressive speedup in relation to the number of active function units in a single-ring system. Second, it has been established that the crude measure, S1/Sinf, of program parallelism is in practice a useful indicator of the suitability of a program for the architecture, regardless of the time variance of the parallelism. Third, a weakness in the present pipeline implementation has been identified, the rectification of which provides better speedup characteristics. Fourth, the effectiveness of the supporting software system has been improved by studying the ratio of instructions executed for each useful floating-point operation in certain large computations. Finally, the need for a Structure Store Unit has been established and specifications for its design have been determined.

It is important, however, to note that this is only an initial attempt at evaluation. In particular, more work is required to determine the behavior of programs that cause matching store overflow. There is also a need to study techniques for parallel algorithm design and transformation and low-level code optimization. In highly parallel programs there is a need to control the amount of active computation by scheduling work within the Token Queue so that matching store requirements are minimized. There also remains the study of multiring systems, in particular the investigatin and evaluation of suitable split functions.

The major long-term interest in dataflow techniques will be in the construction and performance of multiprocessor systems. It is particularly important to know how dataflow systems should be designed for implementation in VLSI, and to be certain that effective software techniques are available for utilizing the hardware. An important advance in this area is the announcement by NEC of a dataflow image-processing chip, the uPD7281 Image Pipeline Processor, which was to be on the market toward the end of 1984.

Acknowledgments. The authors gratefully acknowledge the assistance of their present and former colleagues in the Dataflow Research Group at Manchester, particularly Katsura Kawakami, Adrian Parker, and John Sargeant, who have assisted with the preparation of this paper. Construction of the prototype hardware and software systems has been funded by research grants GR/A/74715, GR/B/40196, and GR/B/74788 from the Science and Engineering Research Council of Great Britain under its Distributed Computing Systems Program. The work has also been supported by an External Research Program grant from Digital Equipment Corporation, the staff of which has written certain of the benchmark programs.

Further Reading. The first four items will serve as useful introductory material for the nonspecialist. On a more specific level, the earliest reference to graphical programming appeared in an obscure internal report within the National Cash Register Corporation in 1958 [5]; a similar idea was published at an MIT Conference in 1962 [6]. The first comprehensive theory for a graphical model of computation, and the most frequently referenced pioneer dataflow paper, was published in 1966 [7]. This was followed by the publication of two influential theses on dataflow computation models at Stanford [8] and MIT [9]. The term "dataflow" was coined in the first of these theses.

There followed a phase of prolific work at MIT by Jack Dennis, who is usually regarded as the instigator of the concepts of dataflow computers as they are now understood. His Computation Structures Group has been responsible for most of the theoretical and development work for static dataflow systems [10–12]. Subsequent static systems have been constructed at CERT-ONERA in Toulouse [13] and Texas Instruments [14].

One of Dennis' early papers [11] suggested the notion of dynamic dataflow. This idea was refined at MIT (in the form of code-copying systems) [15], at Utah by Al Davis [16], and at UCI and MIT by Arvind (in the form of tagged-token systems) [17]. The development of tagged-token dataflow occurred at Manchester simultaneously and independently [18].

The principles of single-assignment programming languages were first published in 1968 [19]. The term "single assignment" was coined by Chamberlin in 1972 [20]. These ideas have subsequently been incorporated into dataflow projects, culminating with the design of the SISAL language in 1983 [21].

REFERENCES

1. IEEE. Special issue on dataflow systems. *IEEE Comput. 15*, 2 (Feb. 1982).
2. Gurd, J.R., Watson, I., Kirkham, C.C., and Glauert, J.R.W. The dataflow approach to parallel computation. In *Distributed Computing*, F.B. Chambers, D.A. Duce, and G.P. Jones, Eds. APIC Studies in Data Processing, vol. 20, Academic Press, New York, Sept. 1984.
3. Glauert, J.R.W. High level languages for dataflow computers. State of the Art Rep. Ser. 10, Number 2, on Programming Technology, Pergamon-Infotech, Maidenhead, U.K., Mar. 1982.
4. Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P. Data-driven and demand-driven computer architecture. *ACM Comput. Surv. 14*, 1 (Mar. 1982), 93–143.
5. Young, J.W., and Kent, H.K. Abstract formulation of data processing problems. Intern. Rep., Product Specifications Dept., The National Cash Register Company, Hawthorne, Calif., 1958.
6. Brown, G.W. A new concept in programming. In *Computers and the World of the Future*, M. Greenberger, Ed. MIT Press, Cambridge, Mass., 1962.
7. Karp, R.M., and Miller, R.E. Properties of a model for parallel computations: Determinacy, termination and queueing. *SIAM J. Appl. Math. 11*, 6 (Nov. 1966), 1390–1411.
8. Adams, D.A. A computational model with data flow sequencing. Ph.D. thesis, TR/CS-117, Dept. of Computer Science, Stanford Univ., Calif., 1968.
9. Rodriguez, J.E. A graph model for parallel computation. Ph.D. thesis, MIT/LCS/TR-64, Laboratory for Computer Science, MIT, Cambridge, Mass., 1969.
10. Dennis, J.B., Fosseen, J.B., and Linderman, J.P. *Data Flow Schemas*. Lecture Notes in Computer Science, vol. 5. Springer-Verlag, New York, 1974.
11. Dennis, J.B. *First Version of a Data Flow Procedure Language*. Lecture Notes in Computer Science, vol. 19. Springer-Verlag, New York, 1974.
12. Dennis, J.B., and Misunas, D.P. A preliminary architecture for a basic data flow architecture. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*. IEEE Press, New York, Jan. 1975, pp. 126–132.
13. Syre, J.C., et al. LAU system—A parallel data-driven software/hardware system based on single-assignment. In *Parallel Computers—Parallel Mathematics*, M. Feilmeier, Ed. Elsevier North-Holland, New York, 1977.
14. Johnson, D. Automatic partitioning of programs in multiprocessor systems. In *Proceedings of the IEEE COMPCON*, IEEE Press, New York, Apr. 1980.
15. Miranker, G.S. Implementation of procedures on a class of data flow processors. In *Proceedings of the IEEE International Conference on Parallel Processing*, IEEE Press, New York, Aug. 1977.
16. Davis, A.L. The architecture and system method of DDM1: A recursively structured data driven machine. In Proceedings of the 5th ACM Symposium on Computer Architecture. *SIGARCH Newsl. 6*, 7 (Apr. 1978), 210–215.
17. Arvind, Gostelow, K.P., and Plouffe, W. An asynchronous programming language and computing machine. Tech. Rep. TR114a, Dept. of Information and Computer Science, Univ. of California, Irvine, Dec. 1978.
18. Gurd, J.R., Watson, I., and Glauert, J.R.W. A multilayered data flow computer architecture. Intern. Rep., Dept. of Computer Science, Univ. of Manchester, England, Jan. 1978.
19. Tesler, L.G. A language design for concurrent processes. In *Proceedings of AFIPS Spring Joint Computer Conference* (Atlantic City, N.J., Apr. 30–May 2). AFIPS Press, Montvale, N.J., 1968, pp. 403–408.
20. Chamberlin, D.D. The "single-assignment" approach to parallel processing. In *Proceedings of AFIPS Fall Joint Computer Conference* (Las Vegas, Nev., Nov. 16–18). AFIPS Press, Montvale, N.J., 1971, pp. 263–270.
21. McGraw, J., et al. SISAL—Streams and iteration in a single-assignment language. Language Reference Manual (version 1.0), Lawrence Livermore National Laboratory, Livermore, Calif., July 1983.

CR Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles; C.4 [**Performance of Systems**]; D.3.2 [**Programming Languages**]: Language Classifications
General Terms: Design, Languages, Performance
Additional Key Words and Phrases: tagged-token dataflow, signal-assignment programming, SISAL

Authors' Present Address: J.R. Gurd, C.C. Kirkham, and I. Watson, Dept. of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, England.