Performance Issues in Dataflow Machines

John GURD, Wim BOHM
and Yong Meng TEO

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, England

Issues affecting the performance of dataflow computers at the machine and language levels are explored. It is suggested that performance is dictated by the nature and the means of identification, distribution and control of workload in the hardware system. Dataflow is an asynchronous concurrent notation based on fine-grain message-passing in graphical programs. Dataflow machines comprise multiple processing elements and structure store modules connected together via a packet-based switching network. Workload is in the form of fine-grain data packets which trigger instruction-level activity in the various components of the hardware architecture. Workload is identified by a compiler for a high-level, single-assignment language, and is distributed across the hardware components dynamically at run-time. The amount of work at any instant can be controlled by a parallelism "throttle". The paper studies the performance of one example of a dataflow computer, the Manchester Dataflow Machine (MDFM).

1. Design Issues for Parallel Computers

1.1. Architectural Levels

This paper explores issues affecting the performance of a particular style of parallel computer system. In considering such issues, it is important to distinguish between different levels of the system architecture.

Broadly speaking, computer systems are designed at three levels. The first, usually thought of as the lowest level, is that of the hardware architec-

ture and its associated machine-code. This is often referred to as the *machine level*. The next, intermediate level is that of the so-called high-level programming language, from which machine-code is generated. This is often referred to as the *language level*. Most often, this is not at a sufficiently high level to express specific applications in a convenient way. Consequently, the highest level, known as the *applications level*, is constructed for particular application areas on top of the language level. The interface to this level is a kind of 'very-high-level', applications-oriented language.

As far as the end-user of a computer system is concerned, performance at the applications level is most important. However, as this varies significantly according to the particular application, and in a somewhat unpredictable fashion, most attempts at evaluating performance are made at the two lower levels. This paper broadly conforms to this pattern, but an attempt is made to relate the performance assessment to a specific set of applications, the so-called *scientific problems*, which involve a substantial amount of numerical calculation.

1.2. Parallel Workload

Parallel system performance is predominantly determined by design decisions made at the machine and language levels. The most critical of these decisions concern the nature, identification, distribution and control of the parallel *workload*.

Different basic models of parallel computation can create workloads of a fundamentally different *nature*. For example, although many parallel programs are written as systems of communicating parallel *processes*, there is a critical distinction between systems that implement the communication using shared memory and those that use direct message-passing. The workload in the former involves accesses to a hardware third party (the shared memory) that are not necessary in the latter. Also, there is an important difference between such *process-based* schemes, which split the overall workload into *coarse-grain* chunks, and systems, such as dataflow, that segregate work into very *fine-grain* components. The former re-

quire infrequent transmission of usually large amounts of data, whereas the latter involve frequent communication of usually small amounts of data.

The *identification* of parallel workload is ultimately the responsibility of the compiler that translates a program from the language level to the machine level. However, certain systems, particularly among those based on dividing the overall problem into coarse-grain processes, require that the programmer be to some extent involved with this process, by *explicitly* identifying those parts of his program that can be executed concurrently. Perhaps a more sophisticated approach is to allow the compiler to derive parallelism *implicitly* from the program. However, conventional programming languages, with their in-built assumptions of sequential execution based on a globally addressable store, and the ability to address individual store locations using distinct program level names (the so-called problem of *aliasing*), often hinder the necessary analysis. Recent developments in language design have addressed this problem by abstracting high-level languages away from the machine level. In principle, the use of program transformation and optimisation techniques to create parallel versions of programs is feasible. In practice, this art is in its infancy, and few such systems are yet in general and reliable use. The identified parallel workload can be of two fundamentally different kinds. In the first, the amount of parallelism is completely determinable in advance of running the program. In the second, the amount of parallelism depends on complex run-time characteristics, and cannot be predicted without running the program.

The problem does not end with the identification of a parallel workload. The different parallel tasks still have to be *distributed* across the various parallel hardware components (processors and memories). Once again, the programmer may have to be involved in the process, explicitly deciding which parallel parts of his program should be executed in which part of the hardware. In making such decisions, the programmer will be concerned to balance the amount of work in distinct hardware components, and to minimise the amount of communication between components, insofar as the actual workload can be predicted. It is already clear that this is very difficult where the number of parallel processes exceeds a few tens and when

the degree of parallelism is unpredictable prior to run-time. Consequently, systems are emerging that make such decisions independently of the programmer. Some such systems determine the distribution of work at compile-time, usually with results similar to those achieved by the programmer. Other systems distribute the workload at run-time, scheduling processes to processors at the time they are created, either *statically*, by a predefined algorithm, or *dynamically*, according to the then current activity level in each part of the hardware. Complex difficulties arise here, due to the need to communicate with other active processes and to minimise the amount of communication traffic in the hardware system.

In many circumstances it is possible for a very high degree of parallel activity to be identified in a program. Where the amount of this software parallelism exceeds the corresponding amount of parallelism available in the hardware, it is usually inefficient to allow all possible concurrent processes to execute. Consequently, there is a need to *control* the amount of active parallelism generated during the program run. This effect has been most noticeable to date in fine-grain parallel systems. However, it is clear that a similar phenomenon is possible in dynamic coarse-grain systems, such as processor 'farms'.

1.3. Absolute and Relative Performance

Traditionally, performance is rated by manufacturers at the machine level, using measurements such as MIPS (millions of instructions per second), usually given as absolute maximum values (guaranteed never to be exceeded). Such rates are *absolute*, in that they express direct hardware performance in machine level terms (instructions are not a recognisable concept at language level).

In order to get closer to the applications level, a measure related to the application area is required. For example, in scientific computations, absolute performance can be measured in MFLOPS (millions of floating-point operations per second). Similarly, symbolic processing rates are often measured in LIPS (logical inferences per second). Such measures do not necessarily correspond directly to hardware activity, since other operations are usually invoked to support these high level actions.

It is often useful to relate machine level activity to these higher level measures, for example by assessing the average number of instructions executed for every floating-point operation, or for every logical inference. Such *relative* measures given an idea of the effectiveness of a particular machine architecture, together with its compilation system, for a particular application area. They are useful because they discount the specific technology used to fabricate a machine.

2. Dataflow System Design

2.1. Introduction

Parallel dataflow systems have been the subject of detailed research since about 1970. Over the past decade, a widespread consensus has been achieved by the dataflow research community concerning the basic underlying model of computation, the organisation of machines, and appropriate high-level programming languages. The principal sites at which practical dataflow research is being conducted, and from which the following presentation is drawn, are the Massachusetts Institutes for Technology (MIT) in the United States of America [4], the ElectroTechnical Laboratory (ETL) in Japan [20,24], and the University of Manchester in England [12,13]. The following sections express the consensus between these groups by inspecting an example of a practical dataflow computing system, the Manchester Dataflow Machine (MDFM), that has been implemented and evaluated at the author's institution. Where consensus does not yet exist, the alternatives under consideration at different sites are presented. The presentation follows the pattern of section 1.2 by concentrating on the nature of the dataflow workload, and the means of its identification, distribution of the dataflow workload, and the means of its identification, distribution across and control within the parallel hardware resources.

2.2. Language Level

At language level, the consensus is to compile code from (usually simple) functional program-

ming languages that are without the sequential and aliasing constraints of conventional languages. Standard functional languages, such as Lisp or ML, are not used because they allow manipulation of store locations in the conventional way. Hence, *pure* functional languages are preferred, but even the commonly used of these, such as Hope and Miranda, are not used by dataflow projects because they are centered on list-structured data, rather than arrays. The style of language most favoured by the dataflow research community is that of the *single-assignment languages*.

Single-assignment languages allow variables to be defined just once in a program, and, as a consequence, they can be read mathematically (as equations) just as easily as they can be read operationally. Examples of dataflow-oriented single-assignment languages are *Val* [1] and *Id* [2], both used at MIT, *DFC* [25], used at ETL, and *SISAL* [16], used at Manchester. *Id* is the most ambitious of these languages, featuring higher-order functions and complex loop structures. *SISAL* is a straightforward, orthogonally designed single-assignment language. *DFC* is still at the conceptual stage of design. It will use the syntax of C, but with single-assignment restrictions, although it is not yet clear how data structures fit into this framework.

SISAL has been designed jointly by researchers from Lawrence Livermore National Laboratory, Digital Equipment Corporation, Colorado State University and the University of Manchester. It was designed to act as a general-purpose non-sequential programming language for experimenting with parallel implementations of various scientific and numerical applications. A front-end compiler converts *SISAL* code into a graphical, machine-independent intermediate format, known as *IF1* [21]. Several standard optimisations, such as eliminating common subexpressions and moving constants out of loops, are performed by *IF1*-to-*IF1* transformations [22]. Machine-specific code generators convert the *IF1* code into appropriate machine-code for a variety of parallel machines, including the Cray-1, the Sequent Balance, and the MDFM.

An example of a compact *SISAL* program is the matrix multiplication program, shown in Fig. 1, which is used to illustrate this paper. Note that Matrix B is in transposed form.

```

export Main
type Vector = array[real]
type Matrix = array[Vector]
function Matmult(A, B: Matrix returns Vector)
  for Row In A
    AnsRow :=
      for Col In B
        Ei :=
          for Rel in Row dot Col In Col
            R := Rel * Col
            returns value of sum R
          end for
        returns array of Ei
      end for
    returns value of concatenate AnsRow
  end for
end function
function Main(N: integer returns array[real])
let
  MRow := array_fill(1, N, 1.0);
  M := array_fill(1, N, MRow);
  PRow := array_fill(1, N, 2.0);
  P := array_fill(1, N, PRow)
in Matmult(M, P)
end let
end function

```

Fig. 1. Matrix multiplication program in SISAL.

2.3. Model of Computation

There are several versions of the dataflow model of computation [3,9,11], but the consensus is to use the *tagged-token* variant. In all dataflow models, machine-code programs are represented as directed graphs in which nodes represent data dependencies. Data is transported along arcs by special vehicles, known as *tokens*, each one of which comes into existence solely for the purpose of carrying one data value from the node that creates it to a subsequent node that needs it. Where a data value is needed by more than one subsequent node, multiple tokens, each carrying the same value, are sent along multiple arcs to the appropriate nodes. In the tagged-token model, each token also carries a *tag* that distinguishes it from every other token on that arc. Tags are generated systematically by specialised instructions so that graphical code can be re-used for cyclic and recursive programs [11]. Tags are also used to distinguish elements of data structures. In the MDFM, a tag has two fields, known as the

activation name (AN—used to separate function calls or loop bodies) and the *index* (IX—used to separate elements of data structures). Nodes are prohibited from generating more than one token per arc with the same tag.

A node is permitted to execute only when a *matching* set of tokens is present on its input arc(s). This criterion is fulfilled when every input arc contains a token carrying the same tag (additional tokens with different tags are ignored). The matching set of tokens is collectively removed from the input arc(s), and the node is executed in such a way that it eventually produces appropriate result tokens on its output arc(s). Conditionals (and cycles) are handled by *branch* nodes with multiple output arcs that receive result tokens selectively, according to some computed condition.

Finally, the consensus is that data structures should be stored in a *data structure store*, and represented by *pointer tokens* when processed in their entirety. Where they need to be processed element-by-element, they can be accessed in the data structure store, either individually or in vector style, using specific *read* and *write* instructions [5,18,19,20].

Figure 2 shows the graphical form of a piece of MDFM machine-code that is typical of tagged-token dataflow machines. It shows optimised code generated for the inner loop of the SISAL matrix multiplication program of Fig. 1. The code in Fig. 2 is a detailed version of that given by Böhm and Sargeant in figure 16 of [7]. Each instruction is prefixed by a letter indicating the nature of its output. “N” shows that the instruction has only one logical output value, although this value may be used twice (as in the case of the DUPLICATE instruction, for example). Other instructions have two logical output values, of which either the left (“L”), the right (“R”) or both (“D”) may be selected. The instructions used are as follows:

- DUP = DUPLICATE incoming token (used to make multiple copies of tokens),
- SYN = SYNchronise two tokens together (used to generate a constant token),
- ADR = ADd together two Real-valued tokens,
- MLR = MuLtiply together two Real-valued tokens,
- CEI = Compare (Equal) two Integer-valued tokens (yields a boolean token),

SBI = SuBtract one Integer-valued token from another,
 BRR = BRanch (Repeat-type) according to value of r.h. boolean token,
 BIX = Branch according to value of r.h. boolean token and Increment indeX,

BRV = BRanch according to zero/nonzero Value of r.h. integer token,
 ESP = Extract the Size of a structure from a Pointer token,
 FSS = Fetch Stream of tokens from Structure store,

Fig. 2. MDFM machine-code for inner product of matrix multiplication.

WSS = Write token to Structure Store,
 SXI = Set indeX using r.h. Integer token,
 SAZ = Set Activation name and Zero index,
 SLB = Set LaBel (i.e. change tag),
 PRL = PRoLiferate l.h. token as a stream of
 tokens of length r.h. value,
 RCK = Range ChecK (used to test for incompat-
 ible matrix sizes),
 KIL = KIL1 (i.e. destroy) all incoming tokens.

These and the other instructions in the MDFM instruction-set are described in detail in the Basic Programming Manual for the MDFM [15].

2.4. Machine Architecture

The separation of data structure storage from instruction processing leads to machines with multiple *processing elements* (PEs) and *structure store modules* (SSs), connected via a packet-based *switching network*. In the case of the MDFM, a *global allocator module* (GA), responsible for resource management, is also included, and the whole system is attached to a conventional VAX

11/780 *host computer* (HC). In the case of the MIT machine, structure storage is contained within the PEs, and certain of the PEs are reserved for resource management.

The basic structure of the MDFM is illustrated in Fig. 3. In the PEs, program graphs are represented as lists of the instructions to be performed (the nodes) plus the addresses of their successor nodes (equivalent to the arcs). Each entry in the list corresponds to a conventional computer instruction. For example, Table 1 shows the literal MDFM machine-code corresponding to the graphical form shown in Fig. 2. This code is held in the *instruction store unit*. Each address in the store contains an instruction-code plus, where appropriate, one or two successor-instruction-address(es) (defining the output arc(s)) and/or a literal operand. Each successor-instruction-address specifies a *matching function* that is used when the corresponding output token reaches the appropriate token matching unit [12,15].

Tokens-on-arcs are represented by *token-packets* containing the data, the tag, and a succes-

Table 1
Literal MDFM machine-code corresponding to the graph of Fig. 2

Address	Instruction code	Destination address(es)	Operand	
			(type)	(value)
1 164:	L SAZ	1 190LBY	B	T
1 174:	L SAZ	1 181LBY		
1 175:	L SAZ	1 183LBY	I	0
1 176:	L BRV	1 177LBY		
1 177:	NSBI	1 178LBY	B	F
1 178:	NDUP	1 179RBY		
1 179:	L PRL	1 194REW	I	0
1 180:	L SXI	1 194REW		
1 181:	L ESP	1 186LEW	I	0
1 182:	NFSS	1 189LEW		
1 183:	L ESP	1 186REW	I	0
1 184:	NDUP	1 176LBY		
1 185:	NFSS	1 189REW	I	1
1 186:	NSBI	1 187LBY		
1 187:	NRCK	1 188LBY	R	0.0
1 188:	NKIL			
1 189:	NMLR	1 193REW	I	0
1 190:	RSYN	1 192LEW		
1 191:	NCEI	1 192REW	I	0
1 192:	DBRR	1 193LEW		
1 193:	NADR	1 194LEW	I	0
1 194:	DBIX	1 195LBY		
1 195:	L SXI	1 196LEW	I	0
1 196:	L SLB	1 201LEW		
1 216:	NWSS

Fig. 3. Multi-ring MDFM machine architecture.

sor-instruction-address. The process of grouping together a set of matching input tokens is performed in the *token matching unit*. The tag-plus-successor-instruction-address of each incoming token is compared against a set of *unmatched tokens*. Where a complete set of matching tokens

is found, the set is sent to the instruction store unit to fetch the appropriate instruction.

Once this successor-instruction has been attached to its set of matching input tokens, the entire *executable packet* is sent to the *processing unit* to be executed. The data and tag(s) resulting

from execution are attached to the new successor-instruction-address(es) to form the output token-packet(s). Output token-packets are sent to the token matching unit in a successor PE through the switching network and a *token queue unit*. The latter is provided simply as a buffer to hold excess token-packets when there is too much parallel activity.

A structure store module (SS) responds to four kinds of token-packet that are normally generated by instructions executed in the PEs. The first kind of packet is a *structure request*, which causes the SS to allocate space for a data structure. Request packets for large structures are sent to the global allocator, rather than an individual SS, so that the structure can be stored in *interleaved* fashion across all the structure memory units, to improve access speed [18,19]. In either case, a *pointer token*, representing the structure as a base address and an upper bound, is returned. The second kind of packet is a *write request*, containing a data value together with the address at which it should be written. The address is constructed by the PE using the original pointer token and the required offset within the structure. If the offset exceeds the upper bound, an error is signalled. The third kind of packet is a *read request*, which accesses a previously written element in a similar manner. If a read request arrives at a SS before the corresponding write request, the read request is automatically *deferred* and then processed later, after the appropriate write packet has arrived. The final kind of packet is a *reference count request*, which modifies the reference count associated with each data structure. These packets are issued by the PEs in line with the usual scheme for reference counting garbage collection [18,19].

Apart from handling requests for allocation of large, interleaved data structures, the global allocator is also responsible for issuing unique activation names (for use in tags) and for 'throttling' the amount of parallel activity in the machine. The latter is achieved by controlling the release of new activation names which are requested by *process request* tokens sent from PEs executing GAN (Generate Activation Name) instructions [17].

There is as yet little consensus about the topology that is most suitable for the switching network that interconnects the hardware components. The Manchester team is experimenting with an equal-path-length switch that involves an equal delay for

every packet transferred through it. The ETL team is using an hierarchical switch in which local transfers are delayed less than non-local ones [20,24]. The latter should be more efficient, provided that the language level compiler can plant code that exhibits suitable *locality*.

2.5. Workload Identification

The overall nature of dataflow workload is determined by the computational model and the basic machine architecture, which were outlined in sections 2.3 and 2.4, respectively. It consists of fine-grain (instruction level) activities in

- (a) the PEs, triggered by the matching together of data tokens in the token matching unit,
- (b) the SSs, triggered by structure handling messages generated in the PEs, and
- (c) the global allocator, also triggered by messages from the PEs.

Tokens and other messages are transmitted around the hardware system by data packets.

In the MDFM, the workload is identified by the IF1 code generator. The total amount of processing and message traffic in each part of the system depends on the nature of the program and the value of data presented to it. Multiple activations of instructions are caused by (a) iterative cycling around SISAL for *initial* loops, (b) recursive calls to SISAL *functions*, and (c) execution of MDFM *proliferate-type* instructions, such as FSS (Fetch Stream from Structure store), planted to implement SISAL *for all* constructs. In the first two cases, tokens are distinguished by the AN field of their tags. In the latter case, the IX field is used. Apart from this, (tokens destined for) different instructions are distinguished by the address of the instruction in the instruction store. Examples of total workload in the MDFM are given in section 3.2.

2.6. Workload Distribution

Workload is distributed across the components of the hardware system dynamically at run-time, but different schemes are used in different machines. In the MIT machine and the SIGMA-1, some attempt is made to 'cluster' instructions together in hardware modules, according to rules that maintain *locality* of communication [4,20]. This implies that tokens and instructions belong-

ing to a logical *process* should be located in 'adjacent' hardware modules. The hierarchical nature of the switching network in these machines defines the nature of adjacency that must be exploited by the workload distribution algorithm.

In the MDFM, the switching network is of uniform delay, and no advantage accrues when instructions are clustered. Consequently, a much simpler distribution algorithm can be used. A randomising *hash function* is applied to the tag-plus-successor-instruction-address of every token arriving at the switching network. The resulting hash address determines which of the PEs the token should be sent to. Small structure request messages can be treated similarly, with the hash address determining which of the SSs to use. Structure access messages are directed to an SS according to the way the corresponding structure has been allocated space.

The above scheme has the disadvantage that, where the tag is used to compute the hash function, code must be replicated in several of the PEs. However, it appears that the effectiveness of workload distribution is greatly improved by using the tag. Results showing this are presented in section 3.3. Corresponding results for the SIGMA-1 system, with its hierarchical switching network, are presented in [14].

2.7. Control of Parallelism

All dataflow researchers agree that some form of parallelism control is necessary, but the study of specific mechanisms for this is in its infancy, so there is little scope for consensus. The MIT group has published a scheme for controlling loop parallelism, using a method called *k*-bounding [6]. They also have a general resource management scheme whose precise details are not yet published. The Manchester group is studying a general parallelism control scheme which they call *throttling* [17]. The ETL group have yet to publish anything in this area.

Although it is not yet implemented in hardware, an overall structure for the MDFM throttle unit has been determined and extensively simulated [23]. The scheme relies on the controlled release of tokens belonging to an instance of a loop body or function call according to the level of activity in the machine when the instance is invoked. This is possible because the code planted

for invoking such an instance involves a special instruction (GAN) to generate the new activation name (AN) for tagging tokens belonging to the instance. Instead of simply releasing the new AN on request, the system can choose to hold up this instance, by denying it an immediate new AN and releasing one later, when contention for resources has diminished. This is achieved by executing GAN instructions in the global allocator, which receives messages from elsewhere in the system to keep it informed of the overall level of machine activity. By allowing progress through the computation in a systematic fashion, the maximum amount of storage required can be significantly reduced. Some results demonstrating this are given in section 3.4.

3. Dataflow Machine Performance

3.1. Benchmark Programs

A number of SISAL programs have been written and their performance on the prototype MDFM has been analysed [10]. A small number of these programs have been analysed in greater detail, using a multi-ring MDFM simulator [23]. The following subsections summarise the simulation results. The programs are as follows:

- MM: The matrix multiplication program of figure 1. This features heavy use of 2-d arrays stored in the SS. The program is written to take full advantage of the vectorising optimisations in the SISAL code generator [7]. Input parameter *n* defines the size of the (square) matrices. The program generates its own input matrices before multiplying them together.
- LR: Solves a simple version of Laplace's equation, over a square grid with fixed boundary conditions, using a straightforward relaxation method. Input parameter *n* defines the size of the square grid. The program generates its own initial data before the relaxation is repeated for 5 cycles. (46 lines of SISAL.)
- ES: Solves a 2-d hydrodynamics problem, including a heat conduction phase, over an *n*-by-*n* square grid. This is by far the most

Table 2

Total workload for five SISAL programs

Program/ datasize	Total tokens	% tokens received				% tokens transmitted				S1	Tokens /Instr.	F1	MMR
		PEs	SSs	GA	HC	PEs	SSs	GA	HC				
MM $n=10$	14272	81.1	16.9	1.2	0.7	83.3	15.8	0.9	0.0	7402	1.9	2000	3.7
MM $n=20$	86546	78.1	20.1	0.9	0.5	79.9	19.5	0.5	0.0	39822	2.2	16000	2.5
MM $n=30$	265422	77.3	21.5	0.8	0.3	78.6	21.1	0.4	0.0	115442	2.3	54000	2.1
LR $n=15$	52729	80.2	18.2	1.2	0.4	86.2	13.4	0.4	0.0	29249	1.8	5070	9.7
LR $n=20$	86226	78.9	19.6	1.1	0.5	85.1	14.6	0.3	0.0	46359	1.9	9720	8.0
LR $n=25$	127597	77.9	20.6	1.1	0.5	84.4	15.3	0.3	0.0	67069	1.9	15870	7.7
ES $n=5$	70608	86.5	12.7	0.6	0.2	89.1	10.8	0.1	0.0	41396	1.7	5771	7.2
ES $n=10$	341654	85.5	13.6	0.6	0.2	87.5	12.4	0.1	0.0	191101	1.8	34853	5.5
BI $n=6$	3312	98.8	0.0	1.1	0.0	99.3	0.0	0.6	0.1	2221	1.5	957	2.3
BI $n=8$	13348	98.8	0.0	1.2	0.0	99.3	0.0	0.6	0.0	8941	1.5	3837	2.3
BI $n=10$	53488	98.9	0.0	1.1	0.0	99.4	0.0	0.6	0.0	35821	1.5	15357	2.3
NQ $n=5$	68344	96.5	2.5	1.0	0.0	97.7	1.7	0.6	0.0	47922	1.4	-	-
NQ $n=6$	288455	97.2	1.8	1.0	0.0	98.0	1.4	0.6	0.0	203359	1.4	-	-
NQ $n=7$	1276914	97.1	2.0	0.9	0.0	97.9	1.6	0.5	0.0	897388	1.4	-	-

complex of the test programs. It is a version of the SIMPLE benchmark, originally written in Fortran by staff at Lawrence Livermore National Laboratory [8]. The SISAL code was produced by Jim McGraw and Steve Skedzielewski during a visit to Manchester in 1983. This version is modified slightly to take advantage of a MDFM machine-code optimisation for accessing arrays whose lower bound is unity [7]. A single iteration is simulated. (1510 lines of SISAL).

- BI: Computes the integral of $(x^2 - 6x - 10)$ using a simple repeated binary split with trapezoidal approximation. Input parameter n defines the depth of splitting before the approximation is applied. Features 'regular' divide-and-conquer recursion which requires good control of parallelism. (32 lines of SISAL).
- NQ: Solves the ' n -queens' chess problem (i.e. works out how to place n queens on an n -by- n chess board in such a way that none of the queens checks one another). Features 'irregular' divide-and-conquer recursion. (79 lines of SISAL.)

3.2. Total Workload

The total workload for the five benchmark programs is summarised in Table 2. The principal

workload is expressed in terms of the total number of tokens passing through the switching network. The next eight columns show how that workload is distributed between the different kinds of hardware module. Only one of each type of hardware module is simulated, so that overheads are minimal. The column labelled S1 gives the total number of instructions executed, and the next column indicates the average number of tokens passing through the switching network for every executed instruction. The final two columns, labelled F1 and MMR, show the total number of floating point operations and the ratio of instructions executed per floating point operation [12], respectively.

The values in Table 2 are obtained by a fully timed simulation of the prototype MDFM hardware which contains a single PE and a single SS. The most essential control option (namely throttling [17]) is included. This gives the best performance on the MDFM with respect to store use [23].

3.3. Balance between Processing and Structure Accessing

Knowledge of the percentage workloads at the input and output of each kind of hardware module makes it possible to anticipate the worst-case average loadings and, hence, to achieve a balance between the amounts of hardware dedicated to

Table 3
Processing and structure accessing rates for five SISAL programs

Program/ datasize	PEtokens/ SStokens	Program parallelism	PE processing rates			SS processing rates actual
			max.	actual	eff.	
MM $n = 5$	6.3	16.7	2.11	1.71	81%	0.27
MM $n = 10$	4.8	47.4	2.27	2.19	96%	0.46
MM $n = 30$	3.6	117.0	2.31	2.29	99%	0.64
LR $n = 25$	3.8	151.1	2.02	1.76	87%	0.47
ES $n = 10$	6.3	179.9	2.36	2.23	94%	0.36
BI $n = 10$	—	124.8	2.46	2.46	100%	—
NQ $n = 7$	48.4	159.7	2.36	2.36	100%	0.05

each function. Clearly, the optimum balance also depends on the relative processing speeds of the various hardware modules. With timing figures obtained from the hardware prototype, the relevant simulation results are summarized in Table 3. The processing rates for each type of hardware module are expressed in millions of input tokens processed per second. The maximum PE processing rates are established by setting all timing delays for the other types of module to zero.

Several versions of program MM are shown first, to illustrate the effect of program parallelism [12]. As the program parallelism increases, the maximum PE processing rate approaches an asymptote determined by the PE pipeline structure and the nature of the MM program. For the other programs, the datasize is selected so that the corresponding asymptote has been reached. As the SS is not pipelined, program parallelism does not affect the SS processing rates in the same way. However, the proportion of different kinds of SS access changes (reads becoming more predomi-

nant as n increases) and this affects the maximum possible processing rate due to the large difference in processing time between reads and the other SS functions.

In most of the programs it is clear that the PEs are working at full rate while the SSs are under-utilised. For the LR and ES programs, structure store accesses are beginning to limit the PE processing rate. This indicates that there should be at least one SS per PE for these programs (or that a faster SS should be constructed). For the other programs it may be possible for one (slow) SS to support more than one PE.

3.4. Effectiveness of Workload Distribution

The effectiveness of the hashing method for distributing workload across multiple PEs and SSs is illustrated in Table 4. The number of SSs is always equal to the number of PEs. For the ES program, there are 4 PEs and 4 SSs. For the other programs, there are 16 PEs and 16 SSs. The

Table 4
Processing and structure accessing rates in a multi-ring MDFM system

Program/ datasize	Tokens		Program parallelism	PE processing rates			SS processing rates actual
	total	%GA		max.	actual	eff.	
BI $n = 8$	13280	0.6	101.6	2.46	1.54	63%	—
BI $n = 10$	53318	0.8	317.0	2.46	2.17	88%	—
BI $n = 12$	213254	0.8	673.0	2.46	2.43	99%	—
BI $n = 12^*$	213318	0.8	121.1	2.46	1.64	67%	—
MM $n = 35$	410590	0.8	498.8	2.31	1.81	78%	0.52
MM $n = 35^*$	409780	0.6	43.9	2.31	0.55	29%	0.16
LR $n = 40$	309601	0.5	290.9	2.02	0.98	49%	0.31
ES $n = 10$	341400	0.5	201.7	2.36	2.13	90%	0.34
NQ $n = 7$	1273984	0.6	390.9	2.36	2.30	98%	0.05

Table 5
Store usage within the PEs with and without the parallelism throttle

Program/ datasize	unthrottled TQ + MS = total	throttled TQ + MS = total
MM $n = 35$	91 911 + 87 122 = 179 033	6764 + 39 500 = 46 264
LR $n = 40$	8185 + 19 793 = 27 978	2225 + 15 015 = 17 240
ES $n = 10$	7665 + 27 282 = 39 947	2564 + 26 803 = 29 367
BI $n = 12$	7781 + 20 941 = 28 722	2819 + 3883 = 6702
NQ $n = 7$	24 545 + 23 446 = 47 991	3593 + 9689 = 13 282

processing rates are quoted per PE. The maximum possible rates are the actual processing rates given in Table 3. The efficiency (eff.) therefore measures the effectiveness of the workload distribution method. The programs marked * use a hash function computed using the successor-instruction-address only.

Several versions of the BI program are shown first, to illustrate the effect of program parallelism. As parallelism increases (with n), the efficiency of PE usage approaches an asymptote determined by the workload distribution. For the other programs, the datasize is chosen so that the corresponding asymptote has been reached. The small increase in total token traffic and the small overall traffic to the GA indicate the low cost of distributing and throttling the parallel workload. The deleterious effect of not using the tag when computing the distribution hash address is apparent.

3.5. Effectiveness of Parallelism Control

The above experiments have all been conducted using the best available control of parallelism [17,23]. The results given in Table 5 show how necessary this is. Here, some of the programs from Table 4 are run with and without the throttle option. The maximum occupancies of the token stores in the token queue (TQ) units and token matching (MS) units in the PEs are shown. Unnecessarily large stores are required unless the throttle is used, except for the ES program, which has an unusual coarse-grain structure and is difficult to throttle in its current form.

As explained in section 2.7, the MDFM throttle works in a coarse-grain fashion, by holding back new ANs for functions and loop bodies when the level of machine activity is too high. As machine activity increases and diminishes, it is important

that the throttle continues to release pending GAN requests at a sensible rate. Hence, a *throttle delay time* (i.e. the minimum possible time between the release of two successive new ANs) is calculated periodically by the throttle and used to limit the rate of release of new ANs. The optimum value of throttle delay time is governed by the time taken by the machine to process all the tokens 'belonging to' a new AN. If ANs are released too quickly, the token stores in the PEs will fill up with excess tokens that will not be processed for a long time. If ANs are released too slowly, the token stores in the PEs will be starved, creating periods of inefficient inactivity. In the experiments cited above, the throttle delay time depends on the instantaneous lengths of the PE token queues and the total number of PEs.

4. Conclusions

We have presented dataflow in terms of the nature of its fine-grain parallel workload, and the means of identification, distribution and control of this workload. By conducting experiments with several benchmark programs, written in the single-assignment language SISAL and run on simulated versions of the Manchester Dataflow Machine (MDFM), we have demonstrated the effectiveness of the dataflow scheme. The total workload, and its distribution across the two major functions of processing and structure accessing, has been assessed. The question of balance between different hardware modules has been addressed. For modestly parallel systems, it has been shown that the workload can be automatically and effectively distributed across multiple hardware modules. We have also demonstrated the feasibility of controlling the proliferation of parallel work in such a system.

These performance characteristics are hard to achieve in a parallel machine design, and, to our knowledge, no other type of system has demonstrated them so convincingly. However, it must be stressed that these ideas are still at an early stage of development and there is much further work to be done. Although the throttle works well for most of the benchmark programs, it behaves badly for the ES program. The irregular size of the processes in this program makes it difficult for the throttle to settle into a steady rhythm. Consequently, large and small processes are released at the same rate, and the workload is not controlled satisfactorily. Also, the paper has failed to consider performance at the applications level, which is surely the most important of all. In particular, there is scope for restructuring the applications themselves in order to exploit hardware parallelism.

Acknowledgements

The authors are indebted to the members of the Dataflow Research Group at the University of Manchester. Dataflow research at Manchester has been supported by the Science and Engineering Research Council of Great Britain, the UK Alvey Programme in advanced information technology, and Digital Equipment Corporation. The SISAL language was designed in collaboration with Lawrence Livermore National Laboratory, Digital Equipment Corporation, Colorado State University and the University of East Anglia.

References

- [1] W.B. Ackerman and J.B. Dennis, VAL: a value-oriented algorithmic language — Reference manual, Technical Report LCS-TR-218, MIT, 1979.
- [2] Arvind, K.P. Gostelow and W. Plouffe, An Asynchronous Programming language and computing machine, Technical Report ICS-TR-114a, UC, Irvine, 1978.
- [3] Arvind and K.P. Gostelow, Some relationships between asynchronous interpreters of a dataflow language, in: E.J. Neuhold, Ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978) 95–119.
- [4] Arvind and V. Kathail, A multiple processor dataflow machine that supports generalised procedures, *Proceedings 8th Annual Symposium on Computer Architecture* (ACM, New York, 1981) 291–302.
- [5] Arvind and R.E. Thomas, I-structures — An efficient data type for functional languages, Technical Memo LCS-TM-178, MIT, New York, 1981.
- [6] Arvind and D.E. Culler, Managing resources in a parallel machine, in: J.V. Woods, Ed., *Fifth Generation Computer Architectures* (North-Holland, Amsterdam, 1986) 103–121.
- [7] A.P.W. Böhm and J. Sargeant, Efficient dataflow code generation for SISAL, Technical Report UMCS-85-10-2, University of Manchester, 1985.
- [8] W.P. Crowley, C.P. Hendrickson and T.E. Rudy, The simple code, Technical Report UCID-17715, Lawrence Livermore National Laboratory, 1978.
- [9] J.B. Dennis, First version of a data flow procedure language, in: *Lecture Notes in Computer Science* **19** (1974) 362–376.
- [10] J.F. Foley, Benchmark test evaluation report, Technical Report UMCS-87-11-2, University of Manchester, 1987.
- [11] J.R. Gurd, Fundamentals of dataflow, in: F.B. Chambers, D.A. Duce and G.P. Jones, Eds., *Distributed Computing* (Academic Press, London, 1984) 3–19.
- [12] J.R. Gurd, C.C. Kirkham and I. Watson, The Manchester prototype dataflow computer, *Comm. ACM* **28** (1) (1985) 34–52.
- [13] J.R. Gurd, C.C. Kirkham and A.P.W. Böhm, The Manchester dataflow computing system, in: J.J. Dongarra, Ed., *Experimental Parallel Computing Architectures* (North-Holland, Amsterdam, 1987) 177–219.
- [14] K. Hiraki, S. Sekiguchi and T. Shimada, System architecture of a dataflow computer, in: *Proceedings 1987 Region 10 Conference on Computers and Communication: toward 2000* (IEEE, 1987) 1044–1049.
- [15] C.C. Kirkham, The Manchester dataflow system — Basic programming manual, Internal Report UMCS-DF-BPM 6th edition, University of Manchester, 1987.
- [16] J.R. McGraw et al., SISAL — Streams and Iteration in a Single-Assignment Language, Language reference manual, Lawrence Livermore National Laboratory, 1983.
- [17] C.A. Ruggiero and J. Sargeant, Control of parallelism in the Manchester dataflow computer, in: *Lecture Notes in Computer Science* **274** (1987) 1–15.
- [18] J. Sargeant, Efficient stored data structures for dataflow computing, Ph.D. Thesis UMCS-85-8-2, University of Manchester, 1985.
- [19] J. Sargeant and C.C. Kirkham, Stored data structures on the Manchester dataflow machine, *Proceedings 13th Annual Symposium on Computer Architecture* (ACM, 1986) 235–242.
- [20] T. Shimada et al., Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations, in: *Proceedings 13th Annual Symposium on Computer Architecture* (ACM, 1986) 226–234.
- [21] S. Skedzielewski and J.R.W. Glauert, IFI — An intermediate form for applicative languages, Reference manual M-170, Lawrence Livermore National Laboratory, 1985.
- [22] S. Skedzielewski and M. Welcome, Data flow graph optimisation in IFI, in: *Lecture Notes in Computer Science* **201** (1986) 17–34.
- [23] Y.M. Teo, Performance evaluation of a heterogeneous multi-ring dataflow machine, M.Sc. Thesis, University of Manchester, 1987.
- [24] T. Yuba et al., SIGMA-1: A dataflow computer for scientific computations, *Comput. Phys. Comm.* **37** (1) (1985) 141–148.
- [25] T. Yuba et al., The SIGMA-1 dataflow computer, Internal report, ElectroTechnical Laboratory, 1987.