Program Partitioning for a Control/Data Driven Computer

Jurij Šilc and Borut Robič

"Jožef Stefan" Institute, Ljubljana, Slovenia

The paper examines the problem of dataflow graph partitioning aiming to improve the efficiency of macro-dataflow computing on a hybrid control/data driven architecture. The partitioning consists of dataflow graph synchronization and scheduling of the synchronous graph. A new scheduling algorithm, called Global Arc Minimization (GAM), is introduced. The performance of the GAM algorithm is evaluated relative to some other known heuristic methods for static scheduling. When interprocessor communication delays are taken into account, the GAM algorithm achieves better performance on the simulated hybrid architecture.

Introduction

There are several commercial and research efforts currently under way to build parallel computers in order to achieve performance far beyond what is possible today. Pipelining and multiprocessing do not break significantly with the standard way in which computers are organized. Processors spend a lot of time fetching instructions and data from memory. On the other hand, *dataflow* computers (DENNIS 1980) depart radically from tradition in order to boost speed enormously. They respond instantaneously to the arrival of data by attaching it to the instruction waiting for it. Instead of fetching the same data several times, copies of data are produced and simultaneously sent to the instructions waiting for it.

In the past 15 years quite a few projects have embarked on the design for a dataflow machine. For a survey of design published before 1986 see (SRINI 1986). Many of these early projects did not get past the design stage and only two were bold enough to announce that their goal was to produce a commercial dataflow machine: the static dataflow project at Hughes Aircraft Co. (VEDDER et al. 1985) and the Data-Driven Signal Processor Project at ESL Inc. (HOGENAUER 1982). Due to various reasons neither of them reached the commercial stage. More successful was the group at NEC, producing μPD7281 – a one-chip processing element based on dataflow. This image pipelined processor did reach the commercial stage and has been available since 1985 (JEFFERY 1985).

Until the mid-eighties the primary goal had been to explore the dataflow approach, but in the last couple of years the emphasis has shifted toward building practical machines. The Dutch company DTN developed a simple dataflow machine based on the μPD7281 (VEEN and van der BORN 1990). In Japan two follow-up dataflow projects are under way. Building on experience with the SIGMA-1, the EM-4 with target structure of more than 1000 processing elements is being produced (SAKAI et al. 1989). The work on the Q-p project has led to the fabrication of a set of five VLSI chips, to be combined next year into a single-chip stand-alone data-driven processor (NISHIKAWA et al. 1987). The seminal dataflow research by Arvind at MIT has recently resulted in an arrangement with Motorola to build a machine based on their Monsoon architecture (ARVIND and NIKHIL 1990).

Rather than adopting dataflow scheduling at the individual instruction level (*fine-grain* dataflow), we consider larger chunks of instructions. Such *large-grain* approach to dataflow by using a hybrid computation model has been termed *macro-dataflow*. In particular, we describe an architecture, named MADAME (MAcro DAtaflow

MachinE), for supporting control/data driven computation. The machine belongs to the family of *hybrid* computer architectures. MADAME is characterized by many processors and a memory/control unit which gathers the results required by subsequent instructions, generates new tasks, and passes them back to processors. The basic idea of MADAME is a result of our previous research of synchronous dataflow architecture (ROBIČ et al. 1987, ŠILC and ROBIČ 1988, ŠILC and ROBIČ 1989, ŠILC et al. 1990). In this paper we consider the problem of optimal construction of instruction chunks aiming to minimize interprocessor communication.

The paper is organized as follows. After a brief description of the organization of MADAME, we introduce a new program graph partitioning method consisting of graph synchronization and scheduling. The partitioning method is illustrated on the FFT algorithm where a considerable speedup improvement was achieved when compared with some well-known scheduling algorithms.

Machine organization

MADAME is a ring consisting of p identical dataflow processors (DFP) and a control unit named Token Flow Manager (TFM). MADAME communicates with the host via the TFM unit (Fig.1).

Though some other processor network might be more suitable we decided to use the ring organization since it offers the possibility of using existing dataflow processors. Having minimum amount of interface hardware used for cascading, an appropriate candidate is μPD7281.

Operation principles

A high-level dataflow program is translated into a machine-level program. A mental image of the latter is suggested by representing it as a dataflow graph (DFG) in which each node represents an instruction and each (directed) arc a conceptual medium over which data items flow.

Let p be the number of DFPs. DFG, viewed as a set of instructions, is partitioned into p subsets using *program graph partitioning schemes* which will be described in detail in the next section. Arcs connecting instructions within the same subset are termed *local* while the others are referred to as *global*. A host is used to assign a different DFP to each subset and performs loading of the subset into the assigned DFP. Concurrently, the assignment of subsets is also recorded in TFM. Moreover, TFM records where the input and output arcs reside and also the dependencies between instructions residing in different DFPs, i.e. global arcs.

The execution starts after the TFM unit has supplied the input data (*operand tokens*), which may be absorbed by several DFPs. Inside the DFPs

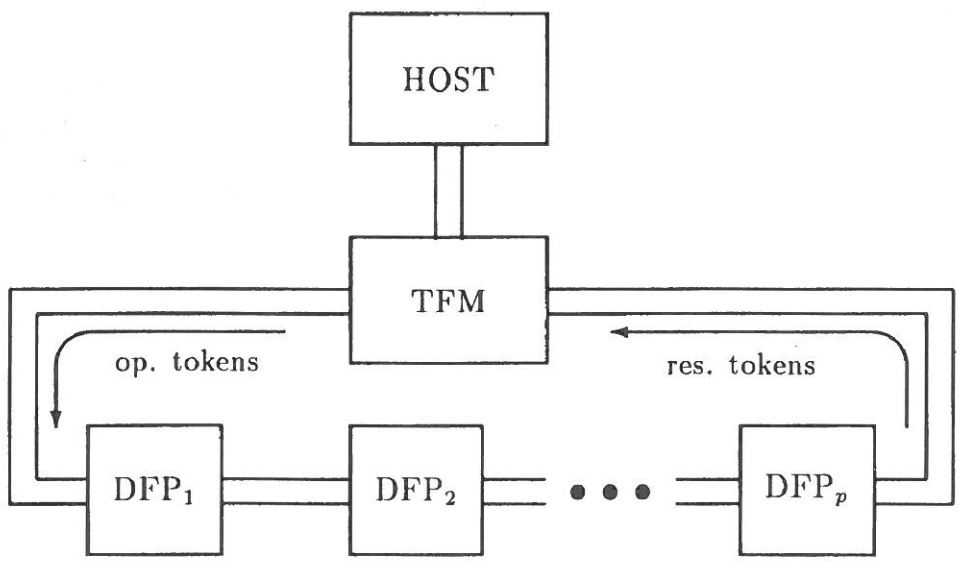

Figure 1: MADAME – MAcro DAtaflow MachinE.

the execution proceeds in an asynchronous fashion according to dataflow principles. However, when data (*result token*) is to be sent to an instruction in some other DFP, it is not sent directly to it as indicated by the corresponding global arc in the DFG. Instead, the result token is first consumed by the TFM unit which, in turn, forwards it as an operand token (with a time delay, if necessary) to the destination DFP. Tokens travelling along global arcs will be termed *global* tokens. Therefore, special attention has to be paid to the instructions (nodes) which are destinations of global arcs, since the operand tokens which fire them arrive from TFM. Consequently, it is the responsibility of TFM to supply these tokens in time so that the total execution time T_p of the DFG will be minimized.

Since instructions related to a global arc are assigned to different DFPs, a *communication delay* between DFPs occurs. Therefore, one of the objectives is to keep the number γ of global arcs (which influences the throughput rate in the ring) as low as possible. This clearly depends on the way the instruction set has been partitioned, and a heuristic static scheduling algorithm, called *Global Arc Minimization* (GAM), has been designed for this purpose.

Dataflow processor

DFP may be any data-driven processor, capable of memorizing and executing parts of DFG and of efficient communicating with its environment. It should facilitate cascading with a minimum amount of interface hardware to increase the throughput rate. There are several candidate DFPs (JEFFERY 1985, NISHIKAWA et al. 1987, KOREN and PELED 1987) among which the NEC μPD7281 has been chosen as the most appropriate one.

Token flow manager

If TFM operated simply as a queue, several disadvantages would appear. For example, an operand token at the front of the queue might be sent to an instruction which could not be fired yet. Also, a token might be sent to an instruction which could postpone its firing without affecting T_p. Therefore, we designed TFM to operate as a *list* as follows:

- an operand token is always sent from the front of the list, and
- an incoming result token is inserted into the list according to the associated *pointer*.

In the next section the construction of information needed for efficient functioning of the TFM will be described.

Program Graph Partitioning

Graph partitioning is a procedure which takes a dataflow graph (DFG) and assigns each node (instruction) to an actual DFP. Partitioning attempts to satisfy three conflicting goals:

1. Maximize the parallelism of the execution in the DFG.
2. Minimize processor resources.
3. Minimize interprocessor communication.

The partitioning can be accomplished during either runtime or compile time. The runtime partitioning (*dynamic* allocation) is a costly approach, however, due to severe supervisory overhead. Therefore, it is preferable to use compile time mechanisms wherever possible (BECK et al. 1990). Since a large number of applications in image processing can be represented by acyclic DFGs, compile time partitioning (*static* allocation) is an important alternative. Therefore, our discussion is limited to the compile time partitioning.

The graph partitioning proceeds in two stages:

- dataflow graph synchronization, and
- scheduling of the synchronous dataflow graph.

Synchronization

Let $V = \{1,2,...,n\}$ be a set of instructions to be executed by a set of identical DFPs. The set V is partially ordered by a data dependency relation $\succ$ so that if $u \succ v$ then u must be executed before v can be initiated. The partially ordered set V is described by a finite acyclic directed graph $DFG = (V,A)$, where V is now viewed as a set of vertices and $A = \{(u,v) \in V \times V \mid u \succ v\}$ is a set of arcs representing data dependencies between vertices. Associated with each vertex v is $t(v) \in \mathbb{N}$ representing its execution time. Given "unlimited" number of DFPs, the minimum execu-

tion time of the DFG is denoted by T_∞. When there are only p DFPs, the minimum execution time of the DFG is denoted by T_p and is $T_p \geq T_\infty$. Conversely, given time T where $T \geq T_\infty$, *the minimum number of DFPs needed to execute the DFG in time* T is denoted by p_T.

Given T and p, the synchronization phase consists of a construction of the

function $s : V \to \{0,1,...,T\}$ such that:

- $\forall v \in V : s(v) + t(v) \leq T$.
- $\forall (u, v) \in A : s(u) + t(u) \leq s(v)$.
- $\forall \tau = 0,...,T : \left| \{v \mid s(v) \leq \tau < s(v) + t(v)\} \right| \leq p$.

Given p and T the *synchronized* DFG, denoted by SDFG(p,T), is a DFG where each $v \in V$ is associated with the number $s(v)$. If v is a vertex then $s(v)$ is the *start* time and $f(v) = s(v) + t(v)$ is its *finish* time. Given p DFPs, a natural goal is to construct SDFG(p,T_p). To do this, the *Time Minimization* synchronization algorithm is used. On the other hand, when a dedicated architecture is designed for a problem domain it is often possible to estimate T_∞. It is natural to construct SDFG(p_{T_∞},T_∞) in this case, since one of the design goals is to minimize processor resources. This is achieved by the *Processor Minimization* synchronization algorithm. Algorithms for SDFG construction were introduced in (ŠILC and ROBIČ 1988, ŠILC and ROBIČ 1989, ŠILC et al. 1990) and are briefly explained as follows.

Let q denote the number of occupied processors at the moment τ. In the *Time Minimization* algorithm, the number of free processors at that moment is $p-q$. Therefore, at most $p-q$ *ready* instructions can be started. Note that some of ready instructions are *urgent*. When starting ready instructions as many as possible urgent instructions are selected. The selection of ready instructions was performed according to three criteria: random selection, increasing $t(v)$ selection, and decreasing $t(v)$ selection. However, none of these criteria proved to be superior. In the *Processor Minimization* algorithm, the number of free processors at some moment is $LBP-q$, where LBP is a lower bound on the number of DFPs. Since urgent instructions have already been taken into account when LBP was computed, only *deferrable* instructions may be selected (using the same criteria as above).

Time Minimization Synchronization Algorithm

input: DFG, p.

output: SDFG(p,T_p), i.e., $s(v)$, for all $v \in V$.

$\tau := 0$; $T_p := 0$; $q := 0$; $W := V$

repeat

if $q > 0$ **then**

$P_f := \{v \in V \mid f(v) = \tau\}$;

$W := W - P_f$; $q := q - |P_f|$

endif

$P_r := \{v \in V \mid v$ has all input operands$\}$;

% P_r are ready instructions.

% Let $P_r = P_u \cup P_d$, where

% P_u have to be fired at the moment τ

% (urgent instructions)

% and P_d need not to be fired at τ

% (deferrable instructions).

if $q < p$ **then**

if $p-q \leq |P_u|$ **then** Let $P_a \subset P_u$, where $|P_a| \leq p-q$ **else**

if $p-q \geq |P_r|$ **then** $P_a := P_r$ **else**

Let $P_a \subset P_d$, where $|P_a| \leq p - q - -|P_u|$; $P_a := P_a \cup P_u$

endif

$q := q + |P_a|$ % Fire some additiona instructions

endif

forall $v \in P_a$ **do** $s(v) := \tau$ **endforall**

$T_p := \tau$; Increment τ

until $W = 0$;

Processor Minimization Synchronization Algorithm

input: DFG, T_∞.

output: SDFG(p_{T_∞},T_∞), i.e., $s(v)$, for all $v \in V$.

Compute LBP, i.e., a lower bound on the number of DFPs;

$\tau := 0$; $p_{T_\infty} := 0$; $q := 0$

repeat

if $q > 0$ **then**

$P_f := \{v \in V \mid f(v) = \tau\}$;

$q := q - |P_f|$

endif

$P_r := \{v \in V \mid v$ has all input operands$\}$;

% P_r are ready instructions.
% Let $P_r = P_u \cup P_d$, where
% P_u have to be fired at the moment τ
% (urgent instructions)
% and P_d need not to be fired at τ
% (deferrable instructions).
$q := q + |P_u|$ % Fire all urgent instructions
if $q < LBP$ **then**
Let $P_a \subset P_d$, where $|P_a| \leq LBP - q$;
$q := q + |P_a|$ % Fire some additional instructions
endif
forall $v \in P_u \cup P_a$ **do** $s(v) := \tau$ **endforall**
$p_{T_\infty} :=$ max (q, p_{T_∞}); Increment τ
until $\tau = T_\infty$;

Scheduling

After SDFG(p,T) has been constructed, the *processor index* $\pi(v)$, $1 \leq \pi(v) \leq p$ of a host DFP is computed for each vertex $v \in V$. The arcs connecting vertices in different DFPs are termed *global* arcs. Since the communication between vertices residing in different DFPs is a time consuming operation, the goal is to schedule vertices so that the interprocessor communication time is minimized. In order to achieve this, a criterion which keeps the number of global arcs as low as possible (and thus improves the throughput rate) was successfully applied.

We can now outline two versions of the *Global Arc Minimization* scheduling algorithm.

GAM-F: Global Arc Minimization Algorithm (Forward version)

input: SDFG(p,T).

output: $\pi(v)$, for all $v \in V$.

Sort pairs (v,$s(v)$) on $s(v)$ and push them on stack S.

% (v,$s(v)$) with minimal $s(v)$ is on top of S.

for $q := 1$ **to** p **do** $F[q] := 0$ **endfor**

repeat

Pop from S pairs with equal s and store them into W.

$P := \{q \mid F[q] \leq s\}$

forall $v \in W$ **do**

forall $q \in P$ **do**

$c(v, q)$ = number of immediate predecessors of v that have been assigned to q-th DFP.

endforall

endforall

Solve WBM-problem for graph $(W \cup P, W \times P)$.

forall pairs (v, q) which are part of the solution **do**

$F[q] := f(v)$; $\pi(v) := q$

endforall

$W := 0$; $P := 0$

until $S = 0$;

GAM-B: Global Arc Minimization Algorithm (Backward version)

input: SDFG(p,T).

output: $\pi(v)$, for all $v \in V$.

Sort pairs $(v, f(v))$ on $f(v)$ and push them on stack S.

% $(v, f(v))$ with maximal $f(v)$ is on top of S.

for $q := 1$ **to** p **do** $F[q] := T$ **endfor**

repeat

Pop from S pairs with equal f and store them into W.

$P := \{q \mid F[q] \geq f\}$;

forall $v \in W$ **do**

forall $q \in P$ **do**

$c(v, q)$ = number of immediate successors of v that have been assigned to q-th DFP.

endforall

endforall

Solve WBM-problem for graph $(W \cup P, W \times P)$.

forall pairs (v,q) which are part of the solution **do**

$F[q] := s(v)$; $\pi(v) := q$

endforall

$W := 0$; $P := 0$

until $S = 0$;

In both versions the *weighted bipartite matching* (WBM) problem (MEHLHORN 1984) has to be solved. The GAM-F version schedules vertices starting at the input vertices of the SDFG and proceeds towards the output vertices tending to assign a vertex v to DFP which hosts maximum number of v's immediate predecessors. Conversely, the GAM-B starts at the bottom of DFG and tends to assign a vertex v to DFP which hosts maximum number of v's immediate successors.

To conclude, the GAM algorithm assigns the corresponding processor index $\pi(v)$ to each vertex $v \in V$. Thus, the graph partitions and the set of global arcs are implicitly determined.

Global Token Construction

After each $v \in V$ has been associated with the processor index $\pi(v)$, the construction of global tokens may be initiated. Remember that data d which flows through global arc (u,v) is encapsulated in the global token and is controled by TFM. Besides data d the global token contains control fields $\pi(v)$ (calculated by the GAM algorithm) and $\lambda(v)$ which points to a location in the $\pi(v)$-th TFM list where data d is to be inserted. The pointer $\lambda(v)$ strongly depends on the start time $s(v)$ (calculated by synchronization algorithms) and is constructed according to the following rule: a vertex v has pointer $\lambda(v) = i$ if there are exactly i–1 vertices which have the processor index equal to $\pi(v)$ and start time less than $s(v)$.

Performance evaluation

In this section, the performance of the algorithm will be analyzed in the problem of computing 8-point Fast Fourier Transform and compared with three other heuristic algorithms for static scheduling, classical Critical Path Method (CPM) (COFFMAN et al. 1976), Heavy Node First (HNF) (SHIRAZI et al. 1990), and Weighted Length (WL) (SHIRAZI et al. 1990).

The DFG in Fig.2 represents the FFT algorithm designed to perform an 8-point FFT transform. The basic operations are addition ⊕ and subtraction-and-multiplication ⊗. The first operation simply adds two input data together, while the latter performs $(I_1 - I_2) \times e^{2\pi ik/8}$, where I_1 and I_2 are two inputs, and $0 \le k \le 7$. The execution time of each ⊕ and ⊗ node is assumed to be one and five time units, respectively. In this case, $T_1 = 72$, $T_\infty = 15$, and $S_\infty = 4.8$, where T_1 is the *sequential* execution time, T_∞ is *minimum parallel* execution time, and $S_\infty = T_1/T_\infty$ is the *ideal* speedup.

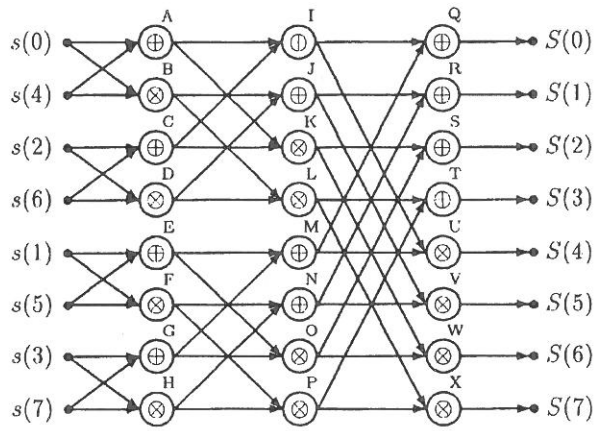

Figure 2: DFG of 8-point FFT

Let T_p denote the minimum parallel execution time when MADAME consists of p DFPs. The corresponding speedup is $S_p = T_1/T_p$. We also define *the degradation* of S_∞ to be $D_p = \frac{S_\infty - S_p}{S_p}$.

The results of the performance analysis are given for $p = 3$ processors and different interprocessor communication delays t_c.

The DFG in Fig.2 is partitioned using schemes CPM, HNF, and WL as shown in Fig.3 (incidentally, all of them give the same result). The dataflow processor P_1 hosts vertices B,H,I,L,Q,R,T,U while D,E,G,K,P,S,X and A,C,F,J,M,N,O,V,W have been assigned to P_2 and P_3, respectively. There are 22 global arcs. For example, global arc (K,W) starts in P_2 at vertex K and ends in P_3 at vertex W. The organization of the corresponding TFM list is given in Fig.4. Note that vertex V is not in the list since it has no global input arcs.

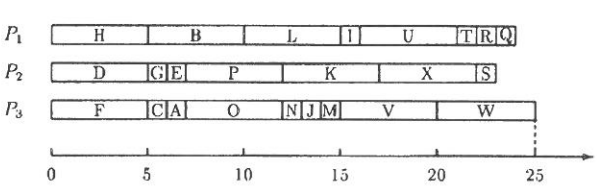

Figure 3: CPM, HNF, and WL scheduling

$\pi(v)$	$\lambda(v)$ 1	2	3	4	5	6	7	8
1	H	B	L	I	U	T	R	Q
2	D	G	E	P	K	X	S	
3	F	C	A	O	N	J	M	W

Figure 4: TFM lists for CPM, HNF, and WL algorithms

Fig.5 and Fig.6 show the results of the scheduling according to the algorithms GAM-B and GAM-F, respectively. Both algorithms radically decrease the number of global arcs compared with CPM, HNF, and WL algorithms, i.e., GAM-B results in 16 and GAM-F in 15 global arcs. This is important when $t_c > 0$ since there are less interprocessor communication delays which in turn improves the speedup (Table 1).

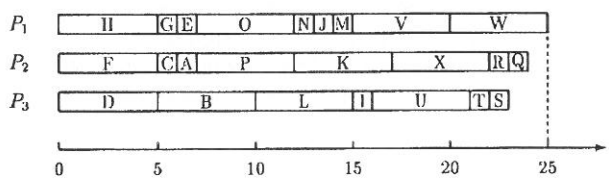

Figure 5: GAM-B scheduling

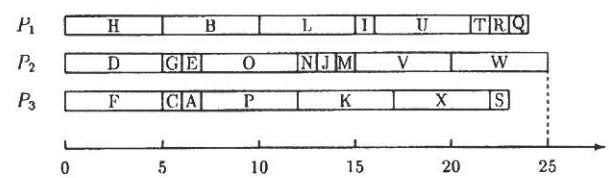

Figure 6: GAM-F scheduling

Table 1: Comparison of the scheduling algorithms.

p=3	CPM, HNF, WL			GAM-F, GAM-B		
t_c	T_p	S_p	D_p	T_p	S_p	D_p
0	25	2.88	0.67	25	2.88	0.67
2	27	2.67	0.80	25	2.88	0.67
4	31	2.32	1.07	28	2.57	0.87
10	43	1.67	1.87	40	1.80	1.67

The TFM lists for the GAM-B and GAM-F algorithms are given in Fig.7 and 8, respectively. Observe that the length of the TFM list was reduced when GAM-F algorithm was applied.

$\pi(v)$	$\lambda(v)$ 1	2	3	4	5	6	7	8
1	H	B	L	I	U	T	R	Q
2	D	G	E	N	J	W		
3	F	C	A	P	X	S		

Figure 7: TFM lists for GAM-B algorithm

$\pi(v)$	$\lambda(v)$ 1	2	3	4	5	6	7	8
1	H	G	E	N	J	W		
2	F	C	A	P	X	R	Q	
3	D	B	I	U	T	S		

Figure 8: TFM lists for GAM-F algorithm

We experimentally found the effectiveness of forward and backward versions of the GAM algorithm to be practically equal. We considered 500 randomly generated graphs with the number n of vertices (instructions) and instruction execution times $t(v)$ also randomly selected ($20 \leq n \leq 120$ and $1 \leq t(v) \leq 10$). Using the *Time Minimization* synchronization algorithm with $p = 1/2p_{T_\infty}$ and having the interprocessor communication delay t_c between 0 and 20, we obtained results depicted in Table 2. Here $D_p^{\uparrow}$ and $D_p^{\downarrow}$ denote degradation of S_∞ resulting from algorithms GAM-F and GAM-B, respectively.

Table 2: Comparison of GAM-B and GAM-F algorithms.

t_c	0	5	10	20
$D_p^{\uparrow}/D_p^{\downarrow}$	1	.9923	.9986	.9935

Using software simulator we also compared the performance of MADAME to pure dataflow architecture (ŠILC 1992). For pure dataflow architecture the instruction start times were computed according to *as-soon-as-possible* firing rule. For MADAME, however, we used the *Time Minimization* synchronization algorithm. In pure dataflow processor scheduling was nondeterministic while in MADAME it was performed according to GAM-B or GAM-F algorithm (whichever was better). Interprocessor communication delays were equal for both architectures.

Special attention was paid to the impact of processor reduction on the total execution time. Generally, when the number of processors is reduced, the speedup is lowered, too. However, this degradation of speedup was less severe in MADAME as described in Table 3.

Table 3: Degradation of ideal speedup.

p	Pure Dataflow	MADAME
	D_p in %	D_p in %
16	0	0
12	1.2	0.2
8	11.9	5.9
4	38.8	21.8

Table 3 shows the results obtained from 100 randomly generated graphs whose maximum parallelism was 16. The number n of vertices (instructions) and instruction execution times $t(v)$ were also randomly selected with $100 \leq n \leq 200$ and $1 \leq t(v) \leq 10$. For example, when only 8 processors were available, pure dataflow architecture exhibited 11.9 % speedup degradation. However, the corresponding degradation on MADAME was only 5.9 %.

Conclusions

Rather than adopting dataflow scheduling at the individual instruction level, larger chunks of instructions were considered (macro-dataflow). The DFG, viewed as a set of instructions, was partitioned into disjoint subsets each of them loaded into its own DFP. Data flow between instructions residing in different DFPs is conceptually described by global arcs. While partitioning the set of instructions of DFG into subsets, two goals have to be achieved. First, the number of the global arcs induced by partition should be minimized. The reason for this is DFP's limited ability to concurrently communicate with its environment, as well as the time delay due to the indirect communication between DFPs. Secondly, subsets should be as large as possible, both to ensure the efficient utilization of DFPs, and to minimize the number of DFPs involved in the computation (ROBIČ et al. 1991). The results of program partitioning would be used by Token Flow Manager during program execution if the MADAME were target architecture.

Acknowledgement

This work was supported by the MZT of the Republic of Slovenia under grant P2-1133-106.

References

ARVIND, R. NIKHIL (1990) Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Computers*, **C-39**, 300--318.

M. BECK et al. (1990) Static Scheduling for Dynamic Dataflow Machine. *Journal of Parallel and Distributed Computing*, **10**(4), 279-288.

E.G. COFFMAN et al. (1976) *Computer and Job-Shop Scheduling Theory*. Wiley-Interscience, New York.

J. DENNIS (1980) Data Flow Supercomputers. *IEEE Computer*, **13**(11), 48-56.

E. HOGENAUER et al. (1982) DDSP – A Data Flow Computer for Signal Processing. Presented at the *Proceedings of the International Conference on Parallel Processing*, Columbus, Ohio.

T. JEFFERY (1985) The μPD7281 Processor. *Byte* (11), 237-246.

I. KOREN, I.PELED (1987) The Concept and Implementation of Data- Driven Processor Arrays. *IEEE Computer*, **20**(7), 102-103.

K. MEHLHORN (1984) *Graph Algorithms and NP-Completeness*. Springer-Verlag, Berlin.

H.NISHIKAWA et al. (1987) Architecture of a One-Chip Data-Driven Processor: Q-p. Presented at the *Proceedings of the International Conference on Parallel Processing*, University Park, Pennsylvania.

B. ROBIČ et al. (1987) Resource Optimization in Parallel Data Driven Architecture. Presented at the *Proceedings of the 5th IASTED Int'l Symposium on Applied Informatics*, Grindelwald, Switzerland.

B.ROBIČ et al. (1991) Graph Compactor for Mapping of Algorithms on VLSI Processor Arrays. Presented at the *Proceedings of the ISMM Int'l Workshop Parallel Computing*, Trani, Italy.

S. SAKAI et al. (1989) An Architecture of a Dataflow Single Chip Processor. Presented at the *Proceedings of the 16th Annual Int'l Symposium on Computer Architecture*, Jerusalem, Israel.

B. SHIRAZI et al. (1990) Analysis and Evaluation of Heuristic Methods for Static Task Scheduling.

Journal of Parallel and Distributed Computing, **10**(3), 222-232.

V. SRINI (1986) An Architectural Comparison of Dataflow Systems. *IEEE Computer*, **19**(3), 68--88.

J. ŠILC, B. ROBIČ (1988) Efficient Dataflow Architecture for Specialized Computations. Presented at the *Proceedings of the 12th World Congress on Scientific Computation*, Paris, France.

J. ŠILC, B. ROBIČ (1989) Synchronous Dataflow-Based Architecture. *Microprocessing and Microprogramming*, **27**(1-5), 315-322.

J. ŠILC et al. (1990) Performance Evaluation of an Extended Static Dataflow Architecture. *Computers and Artificial Intelligence*, **9**(1), 43-60.

J. ŠILC (1992) Time Optimization of Asynchronous Processing with Introduction of Some Synchronization Mechanisms. PhD. Thesis, University of Ljubljana, Slovenia.

R. VEDDER et al. (1985) The Hughes Data Flow Multiprocessor. Presented at the *Proceedings of the 5th Int'l Conference on Distributed Computing Systems*, Denver, Colorado.

A. VEEN, R. van den BORN (1990) The RC Compiler for the DTN Dataflow Computer. *Journal of Parallel and Distributed Computing*, **10** (4), 319-332.

Received: October 19, 1992
Accepted: February 16, 1993

Adress for correspondence:
Jurij Šilc
Jožef Stefan Institute
Laboratory for Computer Architectures
Jamova 39,
61111 Ljubljana
Slovenia
phone: +38 61 159-199
fax: +38 61 161-029
e-mail: jurij.silc@ijs.si

Jurij Šilc received B.Sc, M.Sc and Ph.D degrees in electrical engineering from the University of Ljubljana, Slovenia, in 1979, 1982 and 1992, respectively. In 1980 he joined the Jožef Stefan Institute, Ljubljana and since 1986 he is the head of the Laboratory for Computer Architectures. Between 1980 and 1985 he was working on bubble memory systems and computer networks. His present interests are in parallel processing and computer architectures. He has published over 80 papers and reports.

Borut Robič received B.Sc and M.Sc degrees in computer science from the University of Ljubljana, Slovenia, in 1984 and 1987, respectively. In 1984 he joined the Jožef Stefan Institute, Ljubljana, where he was a research assistant at the Department of Computer Science and Informatics. His main interests are in parallel algorithms, computation theory and combinatorial optimization. At present, Mr. Robič is working on his Ph.D.